UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

TRABAJO FIN DE MÁSTER

Diseño e implementación de un emulador HIL basado en FPGA y Linux

Máster Universitario en Ingeniería de Telecomunicación

Autor: Mario González Ermakov
Tutor: Alberto Sánchez González

Junio, 2021

Diseño e implementación de un emulador HIL basado en FPGA y Linux

AUTOR: Mario González Ermakov
TUTOR: Alberto Sánchez González

Trabajo realizado en el grupo

Hardware & Control Technology Laboratory

Dpto. Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio de 2021

Resumen

La utilización del control digital frente al analógico es cada vez mayor debido a las ventajas como la flexibilidad, fiabilidad y robustez con la que dota a los sistemas. Esto permite controlar diseños cada vez más complejos y compuestos de numerosos elementos. El control digital se integra generalmente con plantas de naturaleza analógica, lo que conlleva que sea necesario realizar simulaciones mixtas previas a la prueba del controlador digital implementado junto a la planta real del sistema. Existen herramientas tradicionales de simulación mixta, pero que resultan ser muy lentas en términos de tiempo.

Actualmente, una solución a este problema que está demostrando excelentes resultados es la llamada simulación HIL (*Hardware-in-the-Loop*), consistente en la emulación en tiempo real de la planta a simular por medio de un modelo que se ejecuta en una determinada plataforma hardware. De esta forma, el controlador no distingue si las señales que recibe se corresponden con las de un prototipo final de la planta o con las de la planta emulada.

En este TFM, se diseña un sistema de emulación HIL donde la planta a emular se trata de un convertidor conmutado de topología *fullbridge*. Este sistema se implementa sobre una FPGA (*Field Programmable Gate Array*), que debido a su gran capacidad de procesamiento paralelo hace que sea posible que los pasos de simulación sean realmente reducidos, rondando las centenas de nanosegundo. Es por ello que las FPGAs se han convertido en una plataforma hardware muy utilizada en investigaciones y herramientas comerciales que tratan sobre este tema. Para el modelado de la planta se ha hecho uso del paradigma de programación HLS (*High Level Synthesis*), que permite reducir el tiempo de diseño, pues hace posible la creación de una implementación RTL de la planta a partir de una descripción en lenguaje de alto nivel.

Tras el desarrollo del modelo del convertidor, se instalará un Sistema Operativo basado en Linux en un procesador ARM incluido en el SoC (System on Chip), que también contiene la FPGA, cuyo papel será el de gestionar las comunicaciones entre la FPGA y una aplicación de escritorio. Desde esta aplicación de escritorio es posible configurar el modelo del convertidor y establecer una condición de *trigger* de otro módulo externo que realiza la función de un sencillo osciloscopio digital. Este osciloscopio captura los datos del convertidor de acuerdo a la condición de *trigger*, y son enviados a la aplicación de escritorio donde se representarán gráficamente.

Finalmente se reflejan las distintas pruebas de integración realizadas que verifican el funcionamiento esperado del sistema y el correcto comportamiento de la planta.

Palabras clave

FPGA, HIL, convertidor conmutado, control digital.

Abstract

The use of digital control over analogical control is increasing due to advantages such as flexibility, reliability and robustness with which it provides systems. This allows controlling more and more complex designs made up of many elements. Digital control is generally integrated with plants of an analogical nature, which means that it is necessary to perform mixed simulations before testing the implemented digital controller integrated with the real plant of the system.

Currently, one solution to this problem that is showing excellent results is the HIL simulation (*Hardware-in-the-Loop*), consisting in the real time emulation of the plant to be simulated by means of a model that runs on a certain hardware platform. In this way, the controller doesn´t distinguish whether the signals it receives correspond to those of a final prototype of the plant or those of the emulated plant.

In this Master Thesis, a HIL emulation system is designed, where the plant to be emulated is a fullbridge switching converter. This system is implemented on an FPGA (Field Programmable Gate Array), which due to its great parallel processing capacity makes it possible to reduce the simulation steps, around hundreds of nanoseconds. That is why FPGAs have become a widely used hardware platform in research and commercial tools about this topic. For the modeling of the plant, the HLS (High Level Synthesis) programming paradigm has been used, which allows to reduce design time, since it makes it possible to create an RTL implementation of the plant from a description in high-level language level.

After the development of the power converter model, a Linux-based Operating System will be installed on an ARM processor included in the SoC (System on Chip), which also contains the FPGA, whose role will be to manage communications between the FPGA and a desktop application. From this desktop application is possible to configure the converter model and set a trigger condition of another external module that performs the function of a simple digital oscilloscope. This oscilloscope captures the converter data according to the trigger condition, and they are sent to the desktop application where it will be graphically represented.

Finally, the different integration tests performed that verify the expected operation of the system and the correct behaviour of the plant are shown.

Key words

FPGA, HIL, switching converter, digital control.

Agradecimientos

Gracias a Alberto, mi tutor, por darme la posibilidad de hacer este trabajo con él. Y quiero agradecer enormemente su mayúscula implicación, paciencia y compromiso durante todo el desarrollo del trabajo, que son dignas de admirar.

Gracias a todas las personas que me han acompañado, ayudado o se han tragado mis chapas en algún que otro momento durante estos 6 años de universidad.

Y gracias también a mis padres por ser un apoyo diario, y especialmente a mi padre, por ser el ejemplo de lo que quería llegar a ser, y que a día de hoy estoy orgulloso de haber conseguido.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 Introducción

1.1 Motivación

Con el paso del tiempo, la electrónica de potencia cada vez está siendo más dominada por el control digital. El control digital aporta numerosas ventajas frente al tradicional control analógico, entre las que destacan [1]:

- La flexibilidad debida a que la implementación del control digital se puede llevar a cabo mediante software/hardware reprogramable, lo cual evita un aumento de costes materiales durante el diseño.

- De la anterior ventaja se desprende otra que es la mayor simplicidad de realizar prototipos y por ende, reducir tiempos de diseño.

- Diseños de mayor fiabilidad por el uso de circuitos integrados menos sensibles a factores ambientales que afecten al envejecimiento de los chips y por su menor vulnerabilidad frente a interferencias.

Sin embargo, los convertidores de potencia son analógicos, lo que conlleva ciertos inconvenientes como la limitación en resolución intrínseca de las señales digitales, el mayor coste asociado al hardware de control digital o la necesidad de introducir ADCs (Analog to Digital Converter) que permitan convertir señales entre el dominio analógico y el digital.

Como en todo desarrollo, con objeto de prevenir errores antes de la fabricación de un producto final, es común realizar simulaciones para asegurar el correcto funcionamiento e integración de las distintas partes involucradas en el diseño. En el caso de convertidores de potencia controlados digitalmente, debe existir una manera de simular conjunta y simultáneamente el dominio analógico característico del convertidor, y el dominio digital propio del controlador/regulador. En el mercado existen simuladores software que realizan estas simulaciones mixtas, y que pese a ser funcionales, presentan el inconveniente de que son muy lentos en términos de tiempo. Además, estos simuladores son capaces de entender esquemáticos o códigos en lenguajes como C, VHDL, entre otros. Sin embargo, estarían simulando en todo caso el algoritmo de control sin tener en cuenta posibles errores en la implementación, integración de periféricos, retardos en los datos, comunicaciones y demás elementos que no estarían simulando.

En base a esto, nace un nuevo paradigma de simulación llamado HIL, consistente en modelar digitalmente el convertidor (o de manera más genérica, la planta) del sistema de control, de forma que a ojos del regulador, sea indiferente si el origen de las señales de los sensores provienen de la versión de la implementación final de la planta o de un modelo digital implementado en una plataforma hardware como un microprocesador o una FPGA (*Field Programmable Gate Array*). Así, es posible validar el comportamiento de la versión final del regulador sin la necesidad de poseer el prototipo final de la planta. De esta manera se ahorra en tiempo y costes de diseño y también se gana en seguridad, pues la planta real puede estar manejando una gran cantidad de energía que puede provocar accidentes graves para el controlador o para los operarios en caso de que algo no funcione como debería.

Actualmente, las FPGA son muy utilizadas ya que debido a sus altas capacidades de procesamiento de baja latencia y alto grado de paralelismo es posible realizar pasos de simulación muy pequeños, permitiendo así llevar a cabo simulaciones en tiempo real. En el mundo de los convertidores de potencia, esto se traduce en la posibilidad de simular convertidores que conmuten a mayores frecuencias. Tanto es así, que esta configuración es objeto de estudio en la actualidad por numerosos artículos de investigación.

Continuando con el tema del procesamiento, hay otro factor importante a considerar que es el tipo de representación de los datos. Hay dos grandes tipos de aritmética. Por un lado está la coma fija, la cual es muy rápida y optimizada pero que, sin embargo, complica el diseño al ser necesario saber la posición de la coma en cada momento. Además, no puede adaptarse a cambios grandes de magnitud en las señales. El otro tipo de representación de datos es la coma flotante, la cual es más lenta computacionalmente pero es más rápida de codificar y su exponente se adapta automáticamente para almacenar la magnitud necesaria Por ello, si las prestaciones lo permiten, la coma flotante es preferible en muchas ocasiones, y de hecho es la que se utilizará en este TFM.

1.2 Objetivos

La meta fundamental de este Trabajo Fin de Máster es la de desarrollar un sistema HIL para emular convertidores de potencia conmutados. Como ejemplo, se modelará un convertidor *fullbridge*. Para ello, se establecen una serie de objetivos que definen las características que poseerá dicho sistema:

- La plataforma hardware sobre la que se emularán los convertidores conmutados estará basado en FPGA (*Field Programmable Gate Array*) y la utilización de un formato numérico en coma flotante (IEEE-754).

- Para el desarrollo del convertidor, se pretende explorar la síntesis de alto nivel (HLS), que permite describir circuitos con lenguajes de alto nivel comúnmente utilizados como C o C++, reduciendo el tiempo de desarrollo.

- Para crear un sistema dotado de comunicaciones de alta velocidad, escalable y versátil se utiliza un dispositivo SoC (Zynq-7000 SoC) provisto de una FPGA y un microprocesador ARM embebido sobre el que se ejecuta un Sistema Operativo basado en Linux.

- Se ofrecerá también un osciloscopio digital integrado en el simulador con algunas funciones básicas de *trigger* para la captura de datos, de forma que una aplicación de escritorio pueda mostrar los resultados de la simulación.

- La información de dicho osciloscopio será trasladada mediante técnicas eficientes de traspaso de información, mediante, por ejemplo, DMA (*Direct Memory Access*).

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- **Capítulo 1: Introducción**

  En este capítulo se presenta la motivación de este TFM, ubicando el contexto en el que se desarrolla, que es el del control digital, y donde finalmente se enumeran los objetivos marcados.

- **Capítulo 2: Estado del Arte**

  En el Estado del Arte se hace hincapié en la importancia de los sistemas de emulación HIL en la industria y se exploran distintas soluciones comerciales basadas en este tipo de sistemas.

- **Capítulo 3: Diseño del emulador HIL de un convertidor *fullbridge***

  Se describe cómo es la arquitectura del sistema HIL que se ha implementado en este TFM. Seguidamente se da paso al análisis de extracción de las ecuaciones que modelan el comportamiento de un convertidor *fullbridge* y finalmente se muestra cómo se realiza su implementación mediante síntesis de alto nivel (HLS).

- **Capítulo 4: Desarrollo e Integración**

  Este capítulo se centra en la integración de todos los elementos de la arquitectura presentada en el capítulo anterior, en la instalación en el procesador de la FPGA de un Sistema Operativo basado en Linux, y en programación del SoC, así como de la aplicación de escritorio que se comunicará con la FPGA sobre la que se emula el convertidor *fullbridge*.

- **Capítulo 5: Pruebas y Resultados**

  Se muestran los resultados del sistema mediante la aplicación de escritorio desarrollada con diversos ejemplos de funcionamiento en el que se muestran todas las funcionalidades implementadas relacionadas con la configuración del *fullbridge* y de un osciloscopio digital que sirve

- **Capítulo 6: Conclusiones y Trabajo Futuro**

  Se concluye el trabajo realizando un análisis del trabajo realizado, su papel en el contexto en el que se ubica, y su potencial mediante el trabajo futuro que se puede llevar a cabo a partir de este proyecto.

2 Estado del arte

Actualmente, el uso de sistemas de control digitales, debido a su gran versatilidad, está muy presente en muchos campos de la industria, tales como el de la automoción, el de la aviación, el energético, entre otros. Asimismo, los sistemas de simulación HIL está aumentando cada vez más su presencia para la verificación de sistemas complejos en campos como la electrónica de potencia [2-5] o la que actúa sobre elementos mecánicos [6-9]. Siempre resulta muy recomendable poder realizar la simulación de los distintos elementos que forman un sistema, sobre todo cuando se tornan cada vez más complejos y están compuestos por numerosos elementos. De lo contrario, sería necesario realizar pruebas de integración mediante la fabricación de prototipos, lo que se traduce en un gasto importante de tiempo y dinero.

Como se comentó en el capítulo 1, en un sistema de control para verificar el correcto funcionamiento del controlador, lo ideal sería poder simular el controlador y la planta de manera simultánea y en tiempo real, lo que implicaría encontrar una manera de simular elementos digitales y analógicos al mismo tiempo. Existen simuladores software mixtos que resultan muy lentos en términos de tiempo, por lo que surgen los sistemas de simulación HIL para cubrir esta limitación.

2.1 Sistemas de simulación HIL.

La simulación HIL (*Hardware in the loop*) hace referencia a los métodos de verificación en tiempo real de un controlador, el cual es conectado a un modelo digital de la planta a regular implementada en una plataforma hardware. Es decir, en lugar de conectar al controlador directamente el prototipo de planta final, se conecta un microprocesador, un DSP, una FPGA o cualquier hardware flexible donde sea posible emular la implementación final de la planta. De esta manera, el controlador no sabrá realmente si tiene conectada la planta final o el modelo que la emula.

El esquema básico de un sistema de simulación HIL es el que se muestra en la Figura 2-1. En este caso se muestra que el hardware sobre el que se programa el modelo de la planta es una FPGA. El controlador, cuyo comportamiento se verifica sobre su versión final implementada sobre hardware, espera señales analógicas de la planta, por lo que los resultados digitales de la planta emulada en la FPGA son convertidas al dominio analógico por medio de un DAC. Finalmente, el controlador manda las señales de control que actúan sobre el modelo provocando una nueva salida.

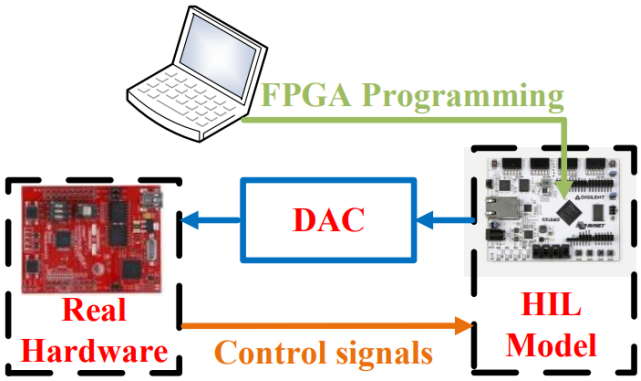

Figura 2-1: Esquema general de un sistema HIL basado en FPGA [20]

En los primeros pasos de este tipo de simulaciones el modelo de la planta se ejecutaba en ordenadores con los que se podían conseguir pasos de simulación del orden de las decenas de microsegundos [10]. Las FPGAs, debido a sus altas capacidades de procesamiento en paralelo, permiten que los pasos de simulación sean muy pequeños (del orden de las centenas de nanosegundo), por lo que son las plataformas más utilizadas actualmente [11-13] para emular plantas que trabajen a frecuencias elevadas, manteniendo así la capacidad de simulación en tiempo real. En el sistema HIL ilustrado en [14], se trabaja con pasos de simulación de 200 ns. Nótese que cuando se habla de tiempo real, realmente se hace referencia a que los datos calculados por la planta estén listos en el momento necesario. Este requisito temporal puede ser muy laxo o muy exigente, por lo que en función de la aplicación se escogerá un hardware u otro en función de las prestaciones necesarias. En HIL se suele entender por tiempo real aquel modelo que permite generar un dato en el momento necesario con un paso de simulación suficientemente pequeño para que la simulación pueda ser precisa. Y para que una simulación sea precisa, el paso de integración debe ser al menos 100 veces más pequeño que la frecuencia de conmutación del controlador [21].

Existen distintas herramientas comerciales que hacen uso de sistemas HIL que toman las ventajas de las FPGAs para ofrecer soluciones en tiempo real en sectores como el de la electrónica de potencia, la automoción o la aviación, entre las que destacan Opal-RT [15], dSpace [16] o Typhoon HIL [17]. Estas herramientas permiten crear modelos complejos de manera gráfica para su posterior simulación. Los pasos de simulación que se pueden conseguir son los mencionados anteriormente (centenas de nanosegundo). Por ejemplo, para el modelo HIL602 de Typhoon HIL, el paso de simulación es de 500 ns. Sin embargo, la principal desventaja de estas herramientas es su elevado coste, que suelen superar la decena de miles de dólares.

Otro aspecto importante a tener en cuenta a la hora de reducir el paso de simulación es el tipo de aritmética que se utiliza en el modelo a implementar. Los dos tipos principales de representación aritmética son la representación en coma fija y la representación en coma flotante.

La coma fija es computacionalmente más rápida y óptima en la utilización de recursos que la coma flotante. En coma fija hay que controlar el ancho de cada variable, que, aunque aumenta el tiempo de diseño, permite aplicar técnicas de optimización para maximizar las

prestaciones, a la vez que la precisión obtenida cumple las especificaciones del sistema [18]. En [19] se hace una comparativa en la utilización de estas aritméticas para modelar convertidores de potencia, donde la coma fija es hasta 10 veces más rápida que la coma flotante. En [20] se ofrecen resultados en FPGAs más modernas de la familia 7 de Xilinx, observándose que la diferencia de pasos de integración entre coma fija y flotante se ha reducido, pero que los pasos de integración siguen siendo mayores para la coma flotante en un factor de 4 o 5 veces. De [21] se extrae una comparativa de los recursos utilizados de una FPGA en el modelado de un convertidor conmutado *fullbridge* en función del tipo de aritmética usada. Los recursos que requiere la coma flotante resultan aproximadamente el doble que los de la coma fija, e incluso son tres veces mayores si se realiza la comparación frente a la forma optimizada de la coma fija.

Sin embargo, la coma flotante es más notablemente más fácil de implementar respecto a la coma fija desde el punto de vista programático, puesto que el diseñador se despreocupa se la posición de la coma en cada operación. Esto implica una reducción del tiempo de diseño. Por ello, para aplicaciones que no requieran una optimización rigurosa del uso de recursos y del tiempo de computación el uso de la coma flotante es más adecuado. Cuando esta optimización es necesaria, sería preferible optar por el uso de la representación en coma fija.

Finalmente, en [20] se muestra una comparativa entre el empleo de coma fija o coma flotante en diferentes paradigmas de programación del modelo (HDL, HLS, HDL con IPs, Matlab, etc…). Los resultados muestran que HLS, aunque ofrece peor rendimiento en algunos tipos de modelos de convertidores, la facilidad de diseño hace que sea un paradigma a tener muy en cuenta, y de hecho, es el que se explora en este trabajo.

3 Diseño del emulador HIL del convertidor fullbridge

En este capítulo se describe en primer lugar cómo es el diseño de la arquitectura que se va a implementar, en segundo lugar se realiza un análisis teórico en el que se extraen las ecuaciones de comportamiento necesarias para un posterior modelado digital de un convertidor *fullbridge*, y finalmente, se expone cómo es la implementación en HLS (*High Language Synthesis*) del convertidor.

3.1 Descripción del sistema

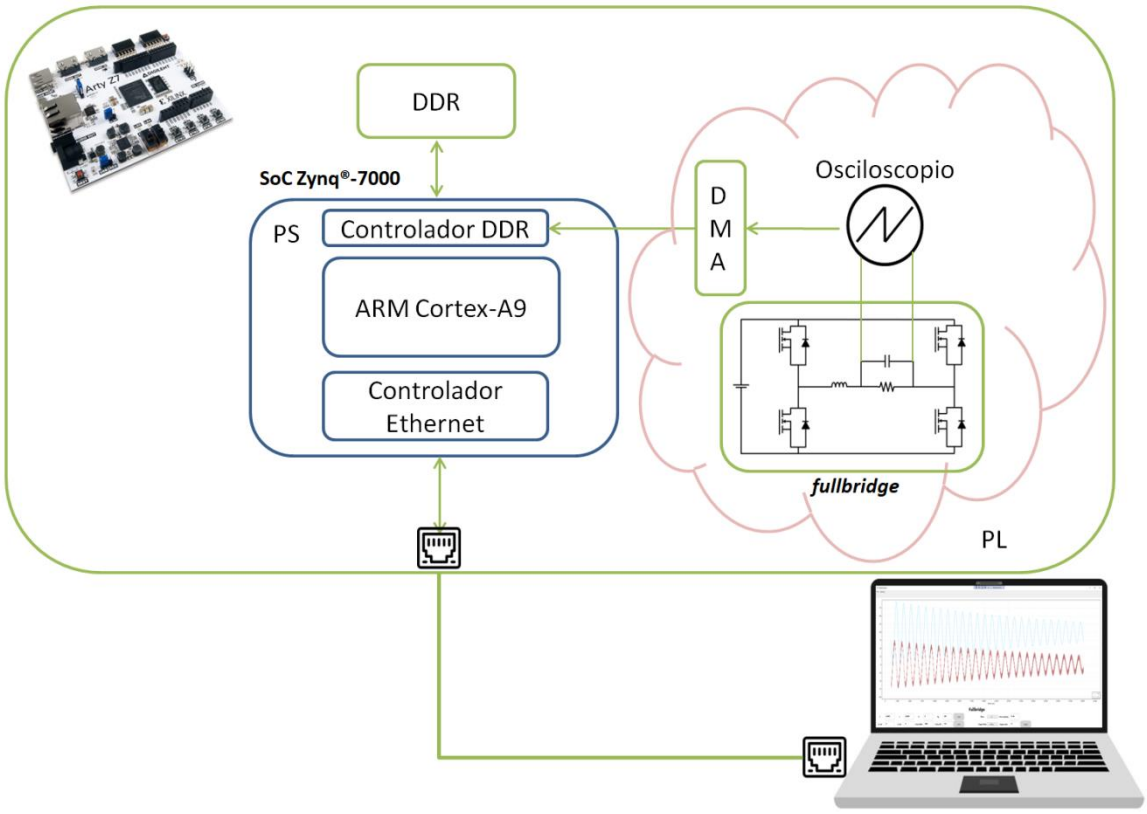

Figura 3-1: Arquitectura principal del sistema

En este Trabajo de fin de Máster, se ha escogido una topología *fullbridge* como ejemplo de convertidor conmutado a modelar. Este modelo es el que va a ser finalmente emulado mediante el hardware en tiempo real. La arquitectura general del sistema responde a la Figura 3-1. El hardware que va a emular el convertidor *fullbridge* en tiempo real es una FPGA (*Field Programmable Gate Array*) de la serie 7 de Xilinx (Zynq-7000 SoC), que además integra un procesador para el desarrollo de aplicaciones basadas en sistemas embebidos. Se escoge una FPGA por su flexibilidad, pues proporciona una plataforma de hardware reprogramable (PL, en color verde), y por su alta capacidad de computación. Además, se trata de una FPGA que posibilita el desarrollo de sistemas embebidos debido a

que cuenta con un procesador de doble núcleo ARM Cortex-A9. Esta funcionalidad se aprovechará para instalar en el sistema de procesamiento (PS, en color azul) un Sistema Operativo basado en Linux desde el cual poder gestionar las comunicaciones con el sistema de emulación y un PC externo que implementará una pequeña aplicación de usuario desde la que controlar el modelo del convertidor, y donde se representarán los datos arrojados por la emulación. Cabe destacar que, el hecho de que esté corriendo un Sistema Operativo basado en Linux en el procesador convierte esta arquitectura en un sistema totalmente escalable, y en este TFM, se ha elegido un pequeño ejemplo práctico y gráfico de su potencial. Por último, en la figura se observa que se añadirá un osciloscopio digital externo gracias al cual se gestionarán los datos que se quieren representar en la aplicación de usuario; y un controlador DMA (*Direct Memory Access*), que hará posible que la ejecución de esta aplicación se lleve a cabo en tiempo real, pues se podrá acceder a la memoria del sistema sin la necesidad de utilizar el procesador activamente para traspasar información entre el modelo del convertidor y la memoria DDR3. Finalmente, se puntualiza que la programación de la FPGA se realizará desde una tarjeta microSD externa.

El primer paso para poder realizar el modelo matemático del convertidor es la extracción de las ecuaciones de un convertidor *fullbridge*.

3.2 Extracción de las ecuaciones de un convertidor fullbridge

Tal y como se observa en la Figura 3-2, se presenta un convertidor *fullbridge* sin pérdidas. Esta topología la conforman: 4 conmutadores, que podrían ser MOSFETs como es en este caso; 4 diodos en antiparalelo, cuya función es la de proporcionar un camino de retorno proveniente de la carga inductiva de la bobina, pues de lo contrario, se generarían picos indeseables de alta tensión; una bobina; un condensador y finalmente una carga que en este ejemplo es puramente resistiva, pero que podría ser de cualquier otro tipo.

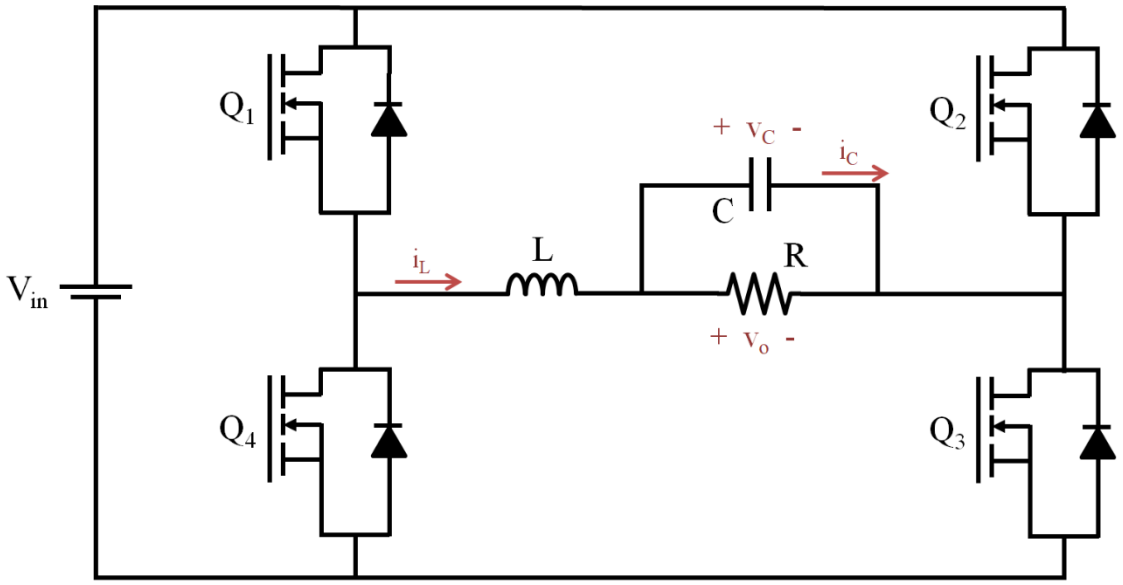

Figura 3-2: Convertidor conmutado, topología *fullbridge*

La topología de la Figura 3-2, puede comportarse de distintas maneras en función de la secuencia de conmutación de los MOSFETs. Una de las formas de operación más usadas con esta topología, es la inversora, es decir, realizando una conversión DC/AC (de corriente continua a corriente alterna). Cabe destacar que, para el modelado de la planta, se debe calcular en cada instante de tiempo el valor de las variables de estado de este modelo [22]. Debido a esto, se debe calcular en cada momento la corriente que atraviesa la bobina, i_L, y la tensión del condensador, v_C, que en este caso coincide con la tensión de salida, v_o, puesto que no se están considerando las pérdidas en el condensador. A continuación, se desarrolla la deducción de las ecuaciones utilizadas para modelar el comportamiento del convertidor conmutado.

Se parte de la tensión que cae en la bobina, v_L

$$v_L = L \cdot \frac{di_L}{dt} \tag{1}$$

Donde L es la inductancia de la bobina y di_L es la variación de corriente de la bobina.

Como se ha comentado, hay que calcular en cada momento la corriente que atraviesa la bobina, para lo cual se transforma la ecuación anterior en una ecuación en diferencias. Existen múltiples métodos para la resolución de ecuaciones diferenciales, entre los que destacan los métodos de Euler explícito e implícito y la familia de métodos Runge-Kutta [23-24]. En este caso las ecuaciones se han resuelto mediante el método de Euler explícito por su baja complejidad y su baja latencia, las cuales son buenas características de cara a poder ser implementado en tiempo real.

$$i_L(k) = i_L(k-1) + \frac{\Delta t}{L} \cdot v_L \tag{2}$$

Donde k hace referencia a cada instante de tiempo, mientras que Δt es el diferencial de tiempo en el que se calcula la variación del valor de las variables de estado. Nótese que en este TFM se utiliza un paso de simulación fijo, que es lo habitual en emulación en tiempo real.

Análogamente, se siguen los mismos pasos para expresar la tensión del condensador v_C, en forma de ecuación diferencial. Esta vez, se parte de la corriente que fluye a través del condensador, de capacitancia C.

$$i_C = C \cdot \frac{dv_C}{dt} \tag{3}$$

Donde dv_C es la variación de tensión del condensador. Transformando en ecuación en diferencias en:

$$v_C(k) = v_C(k-1) + \frac{\Delta t}{C} \cdot iC \tag{4}$$

Para la futura implementación en HLS de las ecuaciones (2) y (4), solo quedan por definir las ecuaciones de v_L e i_C en función de los estados de apertura y cierre de los MOSFETs

del circuito que se expondrán posteriormente. Entonces, resolviendo el circuito para las distintas combinaciones de los MOSFETs, la tensión que cae en la bobina, viene dada por las expresiones de la Tabla 3-1:

Tabla 3-1: Tensión de la bobina, del *fullbridge*. NOTA: Se asume que se trata de un convertidor sin pérdidas.

$v_L =$	$Vin - v_o$	Q1 = ON y Q3 = ON Q1, Q2, Q3, Q4 = OFF y $i_L < 0$ Solo Q1 o Q3 = ON y $i_L < 0$	(5)
	$-Vin - v_o$	Q2 = ON y Q4 = ON Q1, Q2, Q3, Q4 = OFF y $i_L > 0$ Solo Q2 o Q4 = ON y $i_L > 0$	(6)
	$-v_o$	Solo Q1 o Q3 = ON y $i_L > 0$ Solo Q2 o Q4 = ON y $i_L < 0$	(7)

Finalmente, el cálculo de i_C es inmediato conociendo la corriente de la carga R y resolviendo la ecuación del nodo central:

$$i_C = i_L - i_R \tag{8}$$

Como se comentó anteriormente, el modo de actuación de un convertidor conmutado de topología *fullbridge*, puede ser diferente en función de cómo sea la secuencia de conmutación de los MOSFETs. En este TFM, se ha hecho funcionar esta topología como convertidor DC/DC.

El modo de funcionamiento del convertidor consiste en que durante la primera fracción de ciclo de conmutación se cerrarán los MOSFETs Q1 Y Q3, dejando los MOSFETs Q2 y Q4 abiertos, mientras que en la fracción restante del ciclo de conmutación se tendrá justamente lo contrario. El primer estado del circuito se muestra en la Figura 3-3, y el segundo en la Figura 3-4, donde en color verde se resalta la malla en conducción. La señal de conmutación de los MOSFETs, de ciclo de trabajo d, permite conseguir cualquier tensión en bornes del condensador en el rango [-Vg, Vg]. El ciclo de trabajo d, se define como:

$$Ciclo\ de\ trabajo = d = \frac{T_{Q1Q3}}{T_{Q1Q3} + T_{Q2Q4}} \tag{9}$$

Donde T_{Q1Q3} y T_{Q2Q4}, son los tiempos que están en conducción las duplas de MOSFETs (*Q1,Q3*) y (*Q2, Q4*), respectivamente.

La utilización de esta técnica es frecuente para generar una señal de tensión alterna a la salida. Para ello, sería necesario ir variando el ciclo de trabajo de la señal de conmutación según una señal sinusoidal.

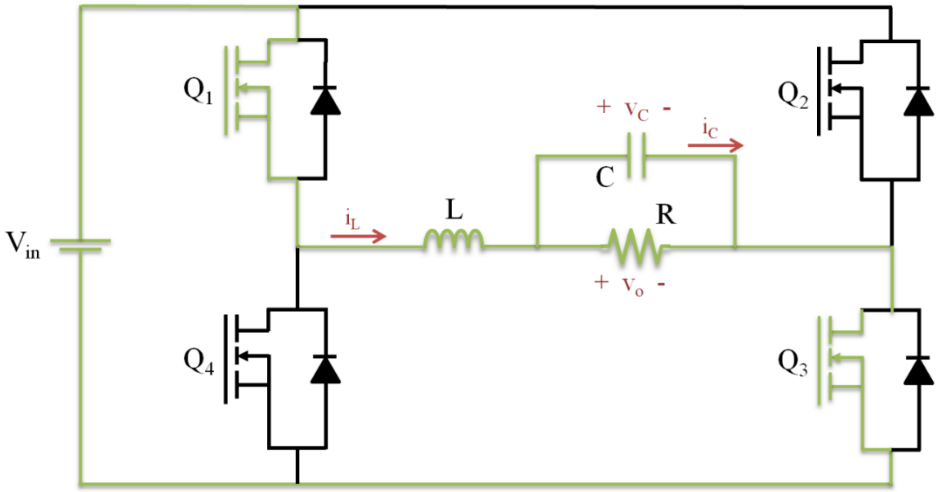

Figura 3-3: Malla 1. Convertidor conmutado, topología *fullbridge*

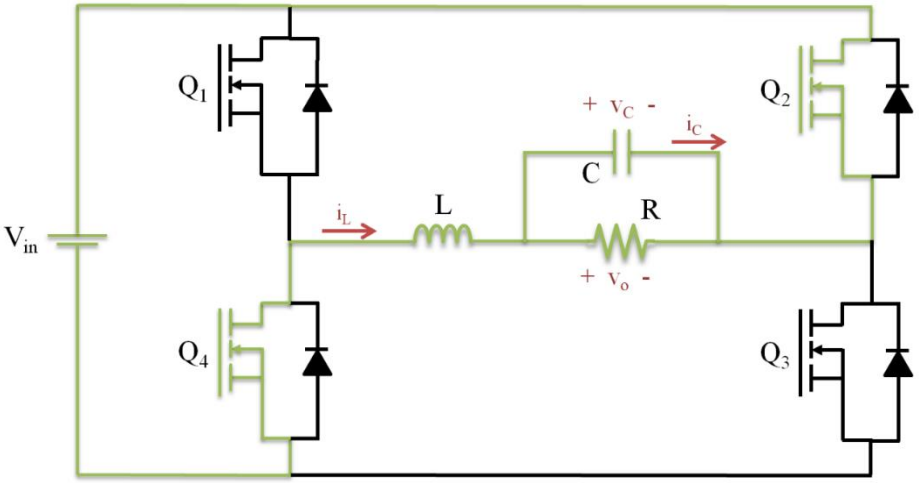

Figura 3-4: Malla 2. Convertidor conmutado, topología *fullbridge*

En la Figura 3-5, se representa el diagrama de las señales de conmutación del circuito, indicando qué conmutadores se mantienen abiertos o cerrados en cada momento.

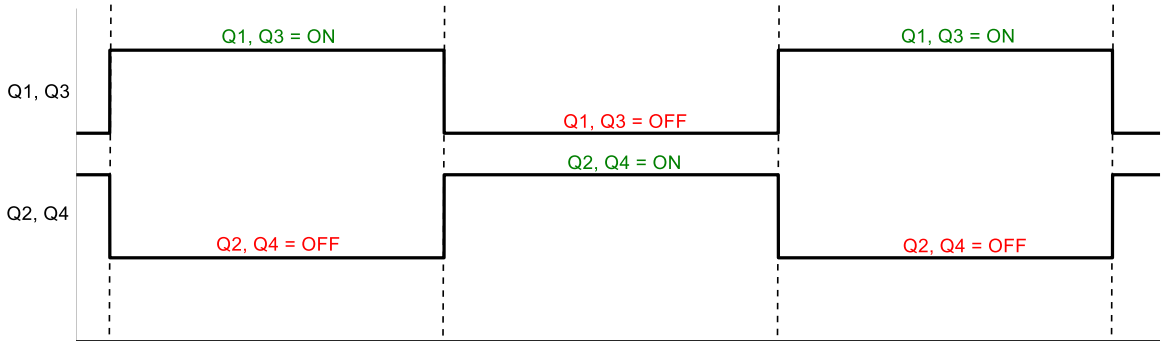

Figura 3-5: Secuencia de conmutación de los MOSFETs

Como se muestra en la Figura 3-6, en la práctica, en la parte de control de los MOSFETs, se puede dejar una pequeña fracción de tiempo con todos los MOSFETs abiertos por una cuestión de asegurar el buen funcionamiento del convertidor, e incluso, dado el caso, su integridad. Esto es debido a que como los MOSFETs no cambian su estado de conmutación de manera instantánea, puede que ocurra que durante algún instante de tiempo dos MOSFETs de una misma rama estuvieran en conducción al mismo tiempo, generándose cortocircuitos, y con ello, mayor consumo, disipación de calor y probablemente la avería del convertidor.

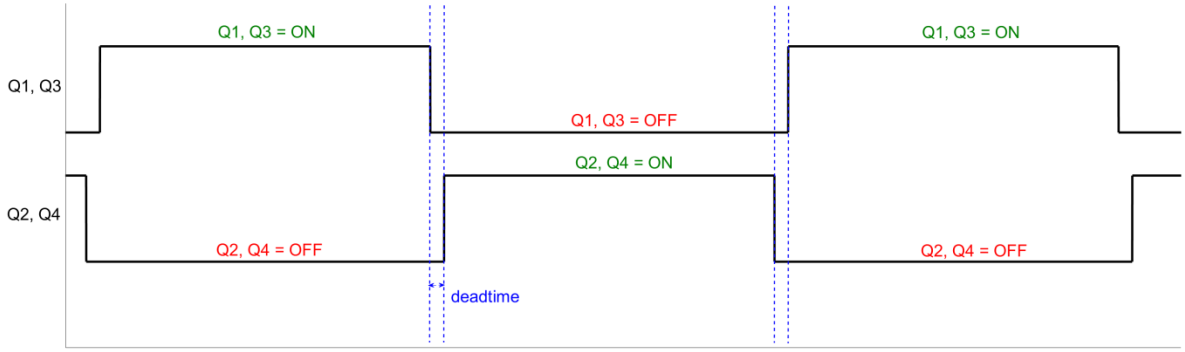

Figura 3-6: Secuencia de conmutación de los MOSFETs, con *deadtime*

Finalmente, aunque no es el objetivo de este trabajo, se puede demostrar que la tensión de salida variará en función del ciclo de trabajo de la señal de conmutación según la siguiente expresión:

$$v_o = V_{in} \cdot (2 \cdot d - 1) \tag{10}$$

Donde V_{in} es la tensión de entrada (tensión de entrada del convertidor). Esto quiere decir que la variabilidad de la tensión de salida v_o, se encuentra en el intervalo $[-V_{in}, V_{in}]$, para un ciclo de trabajo d, comprendido entre 0 y 1, respectivamente.

3.3 Implementación del modelo sintetizable del convertidor fullbridge utilizando HLS

Una vez se ha definido la topología de la planta a simular en la sección 3.2 y se han extraído las ecuaciones que describen su comportamiento, se procede a implementar el modelo con síntesis de alto nivel. Esto implica tanto como su programación como su simulación.

Las técnicas de implementación mediante síntesis de alto nivel (HLS, *High Level Synthesis*) permiten generar una descripción del comportamiento de un hardware en forma de implementación RTL (*Register-Transfer Level*), codificado en un lenguaje HDL (*Hardware Description Language*), a partir de un código realizado en un lenguaje de alto nivel como puede ser C/C++. De esta manera, se toma como ventaja la mayor

productividad que permite alcanzar el diseño software, para así reducir los tiempos de diseño de las distintas partes de un sistema de lógica programable, pues realizar un diseño de cierta complejidad directamente en HDL, resulta más costoso en tiempo y esfuerzo. Las herramientas de HLS generan la implementación RTL de forma automática siguiendo una serie de directivas. Así, es posible tener un código software, y que, sin modificar ninguna línea de código, se puedan producir diferentes soluciones optimizadas de variadas formas en función de los intereses del diseño. Por ejemplo, en un caso puede resultar de interés aplicar directivas de optimización para generar un diseño que reduzca los recursos utilizados de una FPGA, mientras que en otro, se le da prioridad a la latencia de ciertos caminos críticos.

En este TFM, se ha utilizado la herramienta Vivado HLS, que permite realizar las tareas descritas en el párrafo anterior, y por tanto, crear un modelo HDL sintetizado y listo para importarse con posterioridad en el sistema hardware completo que se llevará a cabo en Vivado.

Se han desarrollado dos módulos sintetizables en HLS:

- El propio modelo del convertidor *fullbridge*
- Un modelo encargado de controlar la conmutación de los MOSFETs

A continuación, se describirá a grandes rasgos cómo es la implementación en HLS de ambos módulos.

La cabecera del módulo que implementa el convertidor *fullbridge* se muestra en la Figura 3-7:

```
void fullbridge(bool load, float vc_init, float il_init, float vg, float
ir, bool mosfet_Q1, bool mosfet_Q2, bool mosfet_Q3, bool mosfet_Q4, float
dtL, float dtC, float *vc, float *il, float *numMuestra, float *
vc_debug, float *il_debug);
```

Figura 3-7: Cabecera de la función HLS que implementa el funcionamiento del *fullbridge*

Donde:

- *load*: es un parámetro que funciona a modo en *enable* del módulo. Si es *true*, el convertidor carga los valores iniciales de i_L y v_C (explicados en el siguiente punto) De modo que su utilidad es la de implementar un reset software para inicializar el punto de partida de la simulación.
- *vc_init, il_init*: sirven para ofrecer la posibilidad de empezar la emulación a partir de una cierta tensión inicial en el condensador y una corriente que pasa por la bobina determinadas, en lugar de empezar desde el reposo (*vc_init* = 0, *il_init* = 0).
- *vg*: es la tensión de entrada del convertidor.
- *ir*: es la corriente originada por la carga, que en este caso es R.
- *mosfet_Q1…mosfet_Q4*: son las señales de control de conmutación de los MOSFETs.
- *dtL, dtC*: representan los diferenciales de tiempo $\frac{\Delta t}{L}$ y $\frac{\Delta t}{C}$, de las ecuaciones (2) y (4), respectivamente.

- **vc, *il*: son la tensión en bornes del condensador y la corriente que atraviesa la bobina en cada momento.
- *numMuestra*: variable que representa el último número de muestra calculado en un instante de tiempo. Se le proporciona al osciloscopio digital para tener una referencia de tiempo en cada muestra capturada.
- **vc_debug, *il_debug:* son la tensión en bornes del condensador y la corriente que atraviesa la bobina en cada momento, es decir, contienen la misma información que **vc* e **il*. La diferencia radica en que se envían a través de un puerto AXI_LITE (cuya descripción sucede más adelante), en lugar de tener un puerto físico asignado en la PL. Sirven para depurar las variables de estado mediante el puerto AXI_LITE, sin prestaciones en tiempo real.

Como se ha señalado anteriormente, HLS permite aplicar opcionalmente lo que se conocen como directivas, para optimizar los diseños y ajustar el funcionamiento de los módulos de acuerdo a las necesidades del diseñador. Estas restricciones se llevan a cabo en el código, añadiendo al principio de una línea, el término "#pragma".

```
#pragma HLS interface ap_none port=mosfet_Q1
#pragma HLS interface ap_none port=mosfet_Q2
#pragma HLS interface ap_none port=mosfet_Q3
#pragma HLS interface ap_none port=mosfet_Q4
#pragma HLS interface ap_none port=vc
#pragma HLS interface ap_none port=il
#pragma HLS interface ap_none port=numMuestra

#pragma HLS interface s_axilite port=load bundle=config
#pragma HLS interface s_axilite port=vc_init bundle=config
#pragma HLS interface s_axilite port=il_init bundle=config
#pragma HLS interface s_axilite port=ir bundle=config
#pragma HLS interface s_axilite port=dtL bundle=config
#pragma HLS interface s_axilite port=dtC bundle=config
#pragma HLS interface s_axilite port=vc_debug bundle=config
#pragma HLS interface s_axilite port=il_debug bundle=config
#pragma HLS interface s_axilite port=vg bundle=config
```

Figura 3-8: Directivas HLS utilizadas

En la Figura 3-8, se muestran las directivas utilizadas. Las 7 primeras directivas significan que a las señales especificadas en el parámetro *port*, no se les asignará ninguna interfaz de comunicación, sino que serán puertos directos de entrada/salida, según corresponda. Se corresponderían con los puertos de la entidad de un código en VHDL. Las directivas siguientes, por el contrario, indican que las señales de los puertos indicados pasarán por el procesador Zynq y serán asignadas a una interfaz AXI_LITE, protocolo de comunicaciones maestro-esclavo que se mencionará más adelante.

Para implementar el modelo, hay que traducir las ecuaciones (2) y (4) junto a la Tabla 3-1, dando como resultado el código de la Figura 3-9:

```
if (load == true)
{
      il = il_init;
      vc = vc_init;
}
else
{
      if ((mosfet_Q1 == true && mosfet_Q3 == true) && (mosfet_Q2 == false
&& mosfet_Q4==false))
      {
            vl = vg - vc;
            ic = il - ir;
      }
      else if(){…}

      {…}
}
```

Figura 3-9: Ejemplo de codificación de las expresiones de la Tabla 3-1

Y finalmente, en la Figura 3-10 se van actualizando las ya comentadas variables de estado del convertidor, según las ecuaciones (2) y (4):

```
il = il + vl * dtL;
vc = vc + ic * dtC;
```

Figura 3-10: Ejemplo de codificación de las variables de estado del convertidor

Cabe destacar que se va incrementando una variable con el número de muestra calculado, que será usado, como se explicará más adelante, por el módulo del osciloscopio digital. Para consultar los detalles de la implementación refiérase al Anexo A.

En la Figura 3-11, se muestra la variación temporal de la tensión de salida del convertidor tras realizar la simulación, donde se especifica una tensión del generador de 200 V, y un ciclo de trabajo de conmutación del 75%, por lo que, de acuerdo a la ecuación (10), la salida tenderá a estabilizarse en 100 V. Al no existir ningún tipo de control, las oscilaciones de la figura se deben a la dinámica del convertidor, que responde a un sistema de segundo orden.

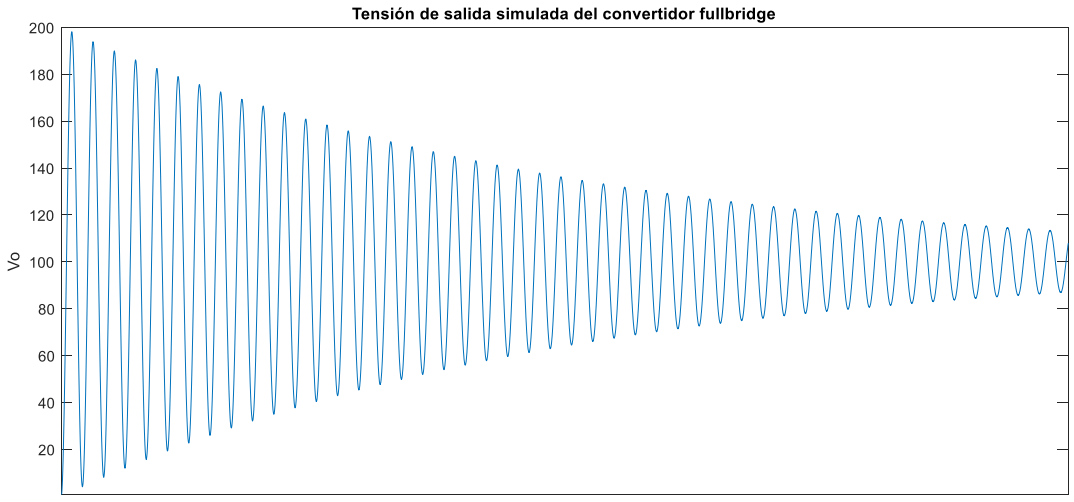

Figura 3-11: Ejemplo de funciones de lectura y escritura de una variable

Por su parte, el módulo de control de la conmutación de los MOSFETs, se encargará de implementar la secuencia de conmutación que generará un cierto valor de tensión a la salida del convertidor. Normalmente, estas señales de control vendrán del exterior, conectándose por tanto a la FPGA que está emulando la planta. Sin embargo, en este TFM se ha generado también un sencillo módulo interno de control de los MOSFETs que genera señales PWM para controlar los cuatro MOSFETs del modelo. Según la Figura 3-12, durante la primera fracción de ciclo de conmutación se cerrarán los MOSFETs Q1 Y Q3, dejando los MOSFETs Q2 y Q4 abiertos, mientras que en la fracción restante del ciclo de conmutación se tendrá justamente lo contrario.

Este módulo toma como entrada *ciclos_on* y *ciclos_max*, que son respectivamente, el número de ciclos de reloj que los MOSFETs Q1 y Q3 estarán en corte, y el número total de ciclos de reloj del sistema que dura una conmutación entera. A partir de estos dos valores, se saca el número de ciclos de reloj que estarán los MOSFETs Q2 y Q4 en corte. Es decir, lo que se está haciendo es configurar el ciclo de trabajo (*duty cycle*) de la señal de conmutación, para que, en función de su valor, se obtenga un valor de tensión u otro a la salida del convertidor *fullbridge*. Trasladado al código, su parte central de funcionamiento se codifica así:

```
if (ciclosActuales < ciclosON)
{
      *salida_Q1 = true;
      *salida_Q3 = true;

      *salida_Q2 = false;
      *salida_Q4 = false;
}
else
{
      *salida_Q1 = false;
      *salida_Q3 = false;

      *salida_Q2 = true;
      *salida_Q4 = true;
}
if (ciclosActuales < ciclosMAX)
      ciclosActuales++;
else
      ciclosActuales = 0;
```

Figura 3-12: Ejemplo de codificación del proceso de conmutación de los MOSFETs

Finalmente, en HLS, se puede hacer una simulación independiente de cada módulo para comprobar que su funcionamiento es el correcto, creando con ese fin, un *testbench* adecuado. Tras ello, se procede a ejecutar la operación de síntesis del modelo, y por último, se empaqueta y se exporta la implementación RTL generada en forma de IP (*Intelectual Property*), para su integración en Vivado. El concepto de IP se define como un conjunto de funciones lógicas preconfiguradas listas para integrarse directamente en un sistema hardware.

4 Desarrollo e integración

Una vez definida la arquitectura del sistema, extraídas las ecuaciones teóricas que modelan el comportamiento de un convertidor *fullbridge*, y realizada su implementación en HLS se procede a su implementación. Antes de proseguir con esta exposición, nótese que la metodología y el entorno de diseño expuestos en los siguientes apartados serían equivalentes independientemente del hardware que se deseara emular.

Las tareas han llevado a cabo en la consecución del objetivo son:

- La integración de la IP generada en el paso anterior, en un diagrama de bloques en Vivado.

- Tomando como entrada el hardware que se exporta de la anterior tarea, implementar la distribución Linux, con ayuda de la herramienta Petalinux. Se genera así una salida consistente en un archivo *booteable* que se utiliza para cargar el programa en la FPGA desde una memoria externa.

- El desarrollo de una aplicación de escritorio que permita configurar y comunicarse con el convertidor *fullbridge*.

- Una vez realizadas y validadas las etapas anteriores para no abarcar todo el diseño en un único esfuerzo y facilitar la depuración del sistema, se procede a la integración de un osciloscopio digital externo, que complementándose con un DMA, terminará de completar toda la funcionalidad de la aplicación de escritorio desarrollada. Ésta consiste en poder visualizar las v_C y i_L, como si de un sencillo osciloscopio se tratase, al que se le puede indicar un nivel de trigger para hacer la captura de las señales. Los pasos a seguir son los mismos que los expuestos en los pasos anteriores: obtención de la IP del osciloscopio digital creada en HLS (en este caso se parte de una ya implementada); integración de la misma en el Block Design junto con el DMA; recompilación de la imagen del SO (Sistema Operativo) que tome el nuevo hardware con las nuevas IP, y modificación de algunas configuraciones del kernel para la instalación de los drivers necesarios para el manejo del DMA

4.1 Integración en un Block Design. Vivado.

Para la integración de todas las partes que componen la Figura 3-1, se han aprovechado las funcionalidades de la herramienta IP Integrator, por medio de la cual es posible llevar a cabo diseños a nivel de bloque de forma gráfica, en concreto con la funcionalidad de *Block Design*. Se consigue así un aumento de productividad ya que:

- facilita los diseños en tanto en cuanto se pueden instanciar los distintos bloques IP y realizar sus conexionados de forma gráfica, evitando tener que hacer lo propio por medio de su correspondiente codificación manual en HDL. Adicionalmente, se posibilita la configuración paramétrica de los IP por medio de formularios.

- hay ciertas tareas que se pueden realizar de forma automática, como el rutado de ciertos puertos presentes en la mayoría de los IP como señales de reloj, de *reset* o interfaces AXI.

AXI es una familia de buses pertenecientes a la especificación AMBA de ARM [25] cuyo funcionamiento está basado en matrices de interconexión punto a punto. De esta manera se introduce en los diseños un bloque llamado *axi_interconnect* encargado de mapear estas conexiones maestro-esclavo. Existen 3 tipos de interfaces AXI:

- AXI4: interfaz con requerimientos de alto rendimiento en la comunicación de dispositivos mapeados en memoria, es decir, en las transacciones de datos se contempla el concepto de leer/escribir datos que se encuentran en una determinada dirección de memoria.

- AXI4-LITE: es una interfaz donde en cada transacción se lee/escribe un solo registro. Esta interfaz es en realidad un subconjunto de la interfaz AXI4 anterior.

- AXI4-Stream: es una interfaz pensada para la transmisión de grandes flujos de datos, o *streams*, a alta velocidad. En estas transacciones no se contempla el uso de direcciones de memoria.

Por las livianas características de comunicación en este sistema, basta con que el tipo de interfaz AXI sea AXI4-LITE, a excepción de la conexión entre el osciloscopio digital y el controlador DMA, y entre el DMA y el controlador DDR3, que se realiza mediante AXI4-Stream. Ello es debido al mayor volumen de datos que se transportan y que se transforman por lo tanto en las conexiones más importantes a tener en cuenta si se desea un sistema de baja latencia.

En la Figura 4-1 se plasma el aspecto que tiene el sistema descrito en la Figura 3-1, en el entorno del *Block Design* de la herramienta IP Integrator.

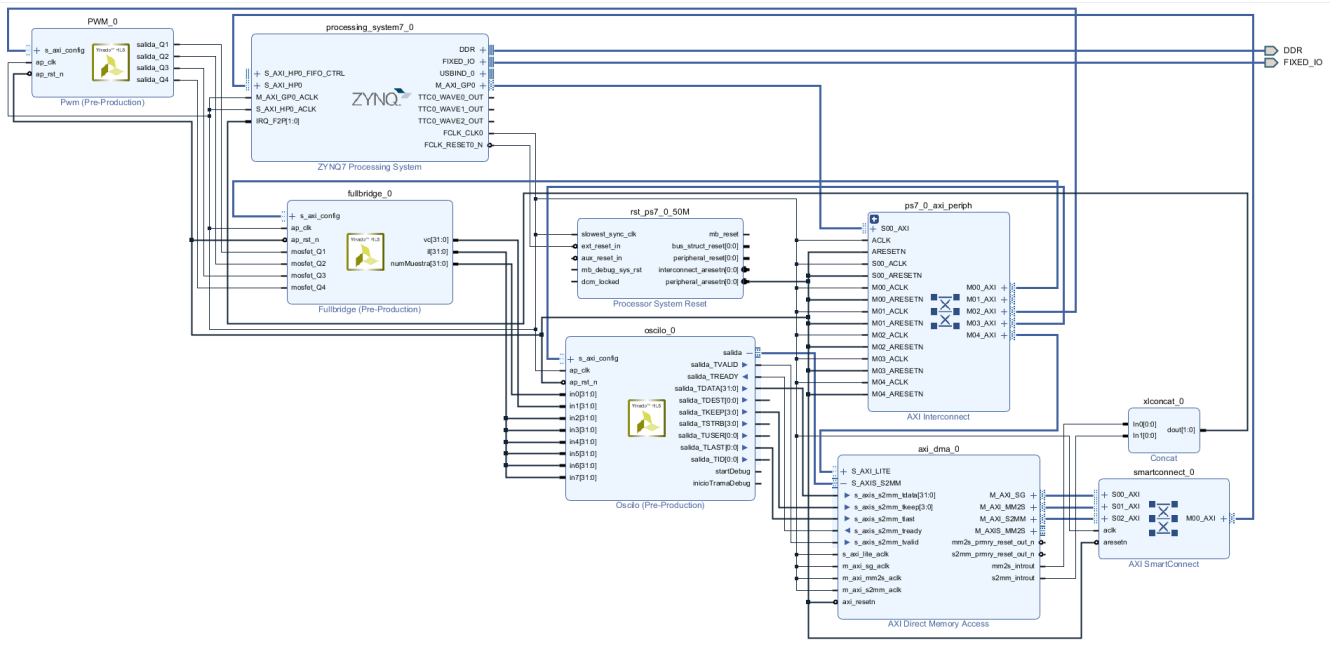

Figura 4-1: *Block Design* realizado en Vivado

Donde se destaca que:

- PWM_0 es el IP de control de los MOSFETs.
- fullbridge_0 es el IP del convertidor *fullbridge*.
- processing_system7_0 hace referencia al procesador.
- ps7_0_axi_periph es la IP que implementa la matriz de conexionado de los buses AXI.
- oscilo_0 es la IP del osciloscopio digital.
- axi_dma_0 es la IP del DMA.

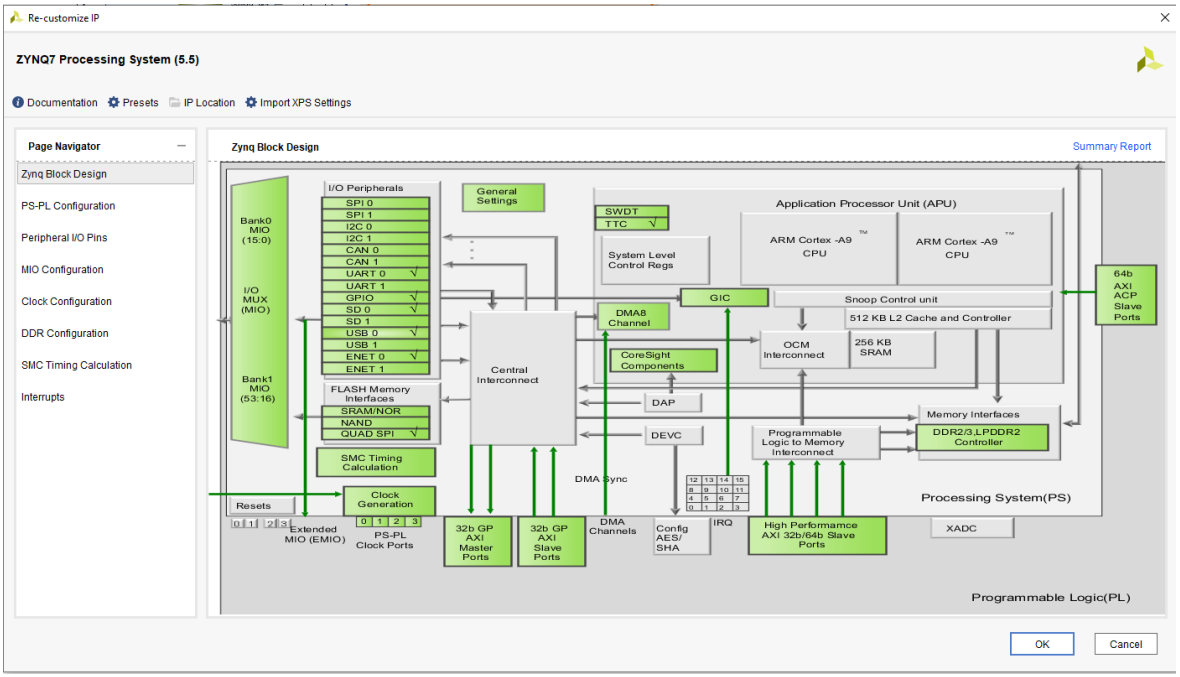

Figura 4-2: Menú de configuración del sistema de procesamiento (PS) ZYNQ7

En la Figura 4-2, se muestra el menú de configuración paramétrica, en este caso, del sistema de procesamiento, donde los recursos que fue necesario habilitar para este TFM son principalmente:

- Un puerto UART, para interactuar con la Shell de Linux mediante una interfaz serie.
- Interfaz de propósito general (GP) maestra para el uso de buses AXI.
- Puerto Ethernet, para la comunicación con otro dispositivo conectado a la red
- USB.
- SD, pues la configuración de la FPGA se cargará desde una memoria SD externa.
- Un TTC (Triple Timer Counter), dado que Linux necesita tener un contador para diversos procesos internos.
- Puerto de alto rendimiento, HP, para el conexionado con el DMA, cuya integración se describirá posteriormente.
- Pin de interrupciones, IRQ_F2P, para permitir la lectura de interrupciones generadas por elementos situados en la lógica programable (PL), concretamente con el DMA.

Una vez finalizado el diseño en Vivado, se realiza el proceso de implementación (asignación de los recursos físicos existentes en la FPGA) y la generación del bitstream (fichero binario que contiene la información del proceso de implementación, y con el que se programa el PL de la FPGA). Por último, se exporta el hardware en formato .hdf, que es un contenedor que contiene el *bitstream* y otros ficheros necesarios para la utilización del hardware en la herramienta SDK, de la cual se hará provecho para la programación del procesador.

4.2 Implementación de la distribución Linux. Petalinux.

Llegado este punto, es preciso recordar que el conjunto de elementos que forman la base del sistema que se está implementando, son la IP del convertidor *fullbridge*, el Sistema Operativo, y la comunicación entre la FPGA y la aplicación de escritorio. Este sistema formaría la base de otras arquitecturas más complejas, como puede ser la integración de un osciloscopio digital como es el caso, para el cual se reserva un apartado. Es por ello que en los subsiguientes apartados se procede a explicar cómo es la implementación de la arquitectura inicial del sistema, que sirve de base para la integración posterior del osciloscopio digital, en cuyo apartado se comentan las modificaciones que hay que realizar en cada uno de los pasos abordados.

Como se ha comentado, la arquitectura Zynq, posee un microprocesador ARM para el desarrollo de aplicaciones embebidas. En este TFM, se ha realizado la instalación y configuración de un SO basado en Linux, llamado Petalinux, el cual es ofrecido por Xilinx. La ventaja de embeber un SO en el procesador es la de poder crear sistemas muy flexibles y escalables. Se permite entonces controlar la lógica de la FPGA mediante Linux, por lo que, a través del mismo, se van a ejecutar las instrucciones necesarias para enviar y recibir datos vía Ethernet entre la FPGA y un PC donde estará corriendo la aplicación de escritorio desarrollada y descrita más adelante. Se tomará ventaja de los buses AXI y del uso del DMA para hacer esta arquitectura más rápida y potente. Aunque estas tareas se podrían realizar mediante un entorno *bare-metal*, y mediante un procesador MicroBlaze implementado en la lógica programable, se ha optado por usar los cores ARM junto a Linux para conseguir la versatilidad, escalabilidad y potencia que eran objetivos de este TFM.

Para realizar la instalación del Sistema Operativo, así como su desarrollo y configuración, se utilizará entorno que ofrece Xilinx con para este propósito. Este entorno cumple su cometido en los sistemas de procesamiento de Xilinx.

Se ha creado una máquina virtual sobre la que se ejecuta Ubuntu 16.04.07 LTS. Tras la instalación de las librerías y dependencias necesarias para la instalación de Petalinux (versión 2018.3, coincidente con la usada en Vivado), reflejadas en su documentación, y tras la creación de un proyecto Petalinux, se han ido ejecutando los pasos siguientes para crear la imagen del sistema operativo que se cargará en una tarjeta microSD externa.

En primer lugar, se tiene que importar la descripción del hardware del sistema, exportado desde Vivado, con el siguiente comando:

```
petalinux-config --get-hw-description
```

A continuación es posible personalizar el sistema operativo a nivel de sistema con:

```
petalinux-config
```

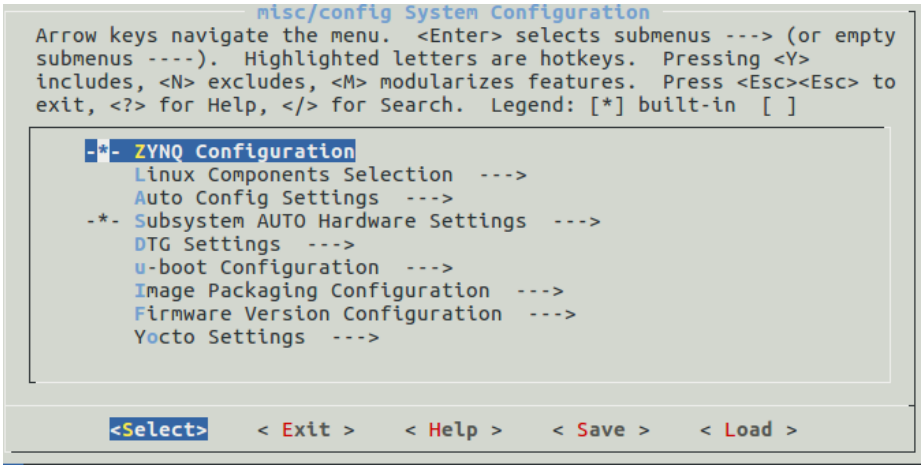

Figura 4-3: Menú de configuración del SO a nivel de sistema

Desde el menú mostrado en la Figura 4-3, se configuran algunos detalles como la dirección IP para poder ser identificado por la red en la que se encuentre para posibilitar las comunicaciones Ethernet, así como el *Gateway* de dicha red.

Con el siguiente comando se realizan configuraciones a nivel de kernel, que en este TFM se modificarán principalmente para posibilitar el uso del driver del DMA como se comentará en el capítulo 4.5.

```
petalinux-config -c kernel
```

Y con el comando de a continuación a nivel de sistema de archivos:

```
petalinux-config -c rootfs
```

Una vez esté configurado todo, se procede a la compilación del sistema operativo con:

```
petalinux-build
```

En este proceso de compilación, se contemplan a grandes rasgos las siguientes acciones:

- Generación del *device tree* cuyo resultado es un archivo DTB, el cual es consultado por el Sistema Operativo en el arranque, y mediante el que conoce la topología y configuración del hardware subyacente.

- Generación del FSBL (*First Stage Boot Loader*), que es la etapa de arranque en la que se carga la imagen del sistema desde memoria no volátil (en este caso, una SD), a memoria volátil (como una DDR).

- Generación del U-Boot, que carga el kernel del SO.

- Generación del kernel de Linux.

- Generación de la imagen del directorio raíz.

Finalmente, para crear el archivo *booteable* para el arranque desde la tarjeta SD externa:

```
petalinux-package --boot --format BIN --force --fsbl
images/linux/zynq_fsbl.elf --fpga images/linux/system.bit --uboot
```

4.3 Programación del procesador ARM

Para la programación del procesador ARM, se ha utilizado la herramienta de Xilinx, SDK (*Software Development Kit*). Consiste es un entorno de desarrollo basado en Eclipse para la programación de aplicaciones embebidas.

En primer lugar, se codifican (en este caso se ha utilizado lenguaje C) las funciones de inicialización de todos los módulos programados, así como las funciones de lectura/escritura de sus parámetros. Para actuar sobre todos esos elementos, es necesario conocer las direcciones base de memoria donde se alojan los registros con la información de cada parámetro. En SDK se crea una plataforma/contenedor hardware a raíz del fichero con la descripción hardware exportada desde Vivado, donde se encuentran ficheros relacionados con el manejo de los módulos. Entre estos ficheros, se localizan los ficheros de cabecera que contienen las direcciones de memoria mencionadas anteriormente. A continuación, en la Figura 4-4, se muestra un ejemplo de inicialización de un módulo:

```
void * fullbridge_init()
{
	unsigned page_size = sysconf(_SC_PAGESIZE);
	unsigned page_addr_fullbridge;
	unsigned page_offset_fullbridge;
	int fd;
	page_addr_fullbridge = (FULLBRIDGE_BASE_ADDR & ~(page_size-1));
	page_offset_fullbridge = FULLBRIDGE_BASE_ADDR - page_addr_fullbridge;

	fd = open("/dev/mem",O_RDWR);
	if (fd < 1) {
		fprintf(stderr,"fullbridge_init: Invalid mem file.\n");
		return NULL;
	}
	void
	*ret=mmap(NULL,page_size,PROT_READ|PROT_WRITE,MAP_SHARED,fd,(FULLBRIDGE_B
	ASE_ADDR & ~(page_size-1)));

	close(fd);

	return ret+page_offset_fullbridge;
}
```

Figura 4-4: Función de inicialización del módulo *fullbridge*

Como se puede apreciar, se calcula la dirección base de memoria de la página en la que se encuentra el módulo, para calcular posteriormente el offset del módulo *fullbridge* en la página en la que se encuentra. Con todo ello, se calcula la variable *ret*, sirviéndose de la

función *mmap*, para realizar un *mapeo* entre la dirección de memoria del periférico visible por el programa y la dirección real de dicho periférico. De esta manera, la función devuelve un puntero con la dirección de memoria base del módulo *fullbridge* que se necesita para hacer referencia a sus parámetros.

Una vez inicializados los módulos, ya se puede hacer uso de sus funciones de lectura/escritura de sus parámetros. A modo de ejemplo, en la Figura 4-5, se muestra la función de lectura y escritura de la variable que representa la tensión inicial del condensador del convertidor. Se devuelve un puntero a la dirección de memoria de cada parámetro que se lee o escribe. Para calcularla, se suma *ptr*, que es el puntero que apunta a la dirección base de memoria del módulo, y el offset del parámetro que se lee o escribe respecto a esa dirección:

```
float FullBridge_Get_VC_INIT(void *ptr)
{
	return *((float *)(ptr+XFULLBRIDGE_CONFIG_ADDR_VC_INIT_DATA));
}

int FullBridge_Set_VC_INIT(void *ptr, float Data)
{
	*((float *)(ptr+XFULLBRIDGE_CONFIG_ADDR_VC_INIT_DATA)) = Data;
}
```

Figura 4-5: Ejemplo de funciones de lectura y escritura de una variable

En esta primera etapa de creación del sistema básico, el flujo del programa consta de dos procesos: un hilo principal (*main*), y un hilo secundario dedicado a la recepción de datos de configuración procedentes de la aplicación de escritorio.

Las principales tareas que realiza el hilo principal son:

- Creación de la figura del servidor, para lo cual se crea un socket TCP/IP que escucha en la dirección IP configurada para la FPGA y en un determinado puerto. Con el socket ya creado, se lanza el hilo secundario dedicado a la escucha de nuevos datos de configuración de llegada.

- Habilitación ordenada de los IP del convertidor *fullbridge* y de control de los MOSFETs.

- En un bucle, realizar una pequeña simulación de lo que sería el sistema completo que incluye el osciloscopio digital, esto es, la preparación de las tres señales que se envían hacia la aplicación de escritorio (v_C, i_L y un vector de tiempos, t) de acuerdo a la condición de trigger que se seleccione desde la aplicación de escritorio. Nótese que esto mismo es parte de lo que realiza el IP del osciloscopio digital y que por tanto será sustituido en el futuro, pero que ha servido de ayuda para mejorar el entendimiento de la funcionalidad final del sistema completo.

Las principales tareas que realiza el hilo secundario son:

- Leer el buffer de recepción de la interfaz Ethernet conectada a la aplicación de escritorio, a la espera de datos.

- Decodificar los datos recepcionados mediante el primer byte leído a fin de discernir entre lo que son datos de configuración del convertidor y datos de configuración de *trigger*, concretados en la siguiente sección.

- Ejecutar las funciones pertinentes de escritura antes indicadas para la actualización de los parámetros de cada IP. En caso de que los datos recibidos sean de configuración del convertidor, se debe resetear el convertidor para su recomienzo con los nuevos valores de sus componentes.

4.4 Programación de la aplicación de escritorio

Tal y como se ha mencionado con anterioridad, y a modo de ejemplo de validación del funcionamiento del sistema, así como de demostración de las posibilidades que ofrece una arquitectura como la desarrollada en este TFM, se ha programado una aplicación de escritorio con dicho fin. La programación se ha llevado a cabo en el entorno de desarrollo integrado de Windows, Visual Studio. Esta aplicación basada en formularios (*Windows Forms*), que forma parte del framework .NET de Microsoft, consiste en una interfaz gráfica que simula un sencillo osciloscopio con el que poder realizar mediciones de la tensión de salida del convertidor, v_C al igual que la corriente que atraviesa la bobina, i_L. Siguiendo la metodología habitual de Windows Forms, las interfaces gráficas se han diseñado en XAML (*eXtensible Application Markup Language*), en un entorno como el de la Figura 4-6, donde en la mitad inferior de la figura se encuentra el editor de XAML, y en la mitad superior el resultado gráfico. Por otra parte, la dinámica y la funcionalidad del programa se ha implementado en C Sharp (C#).

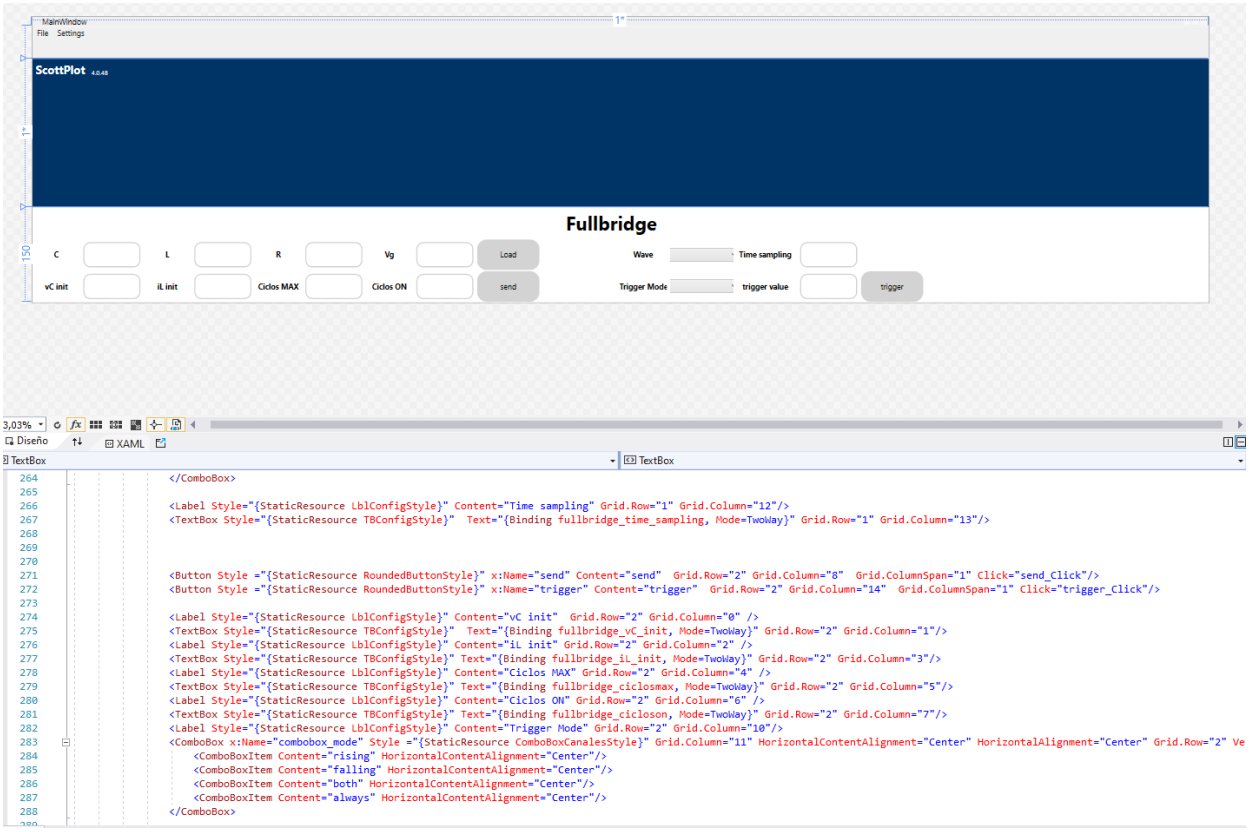

Figura 4-6: Entorno de diseño de la ventana principal, basado en XAML

La aplicación envía datos hacia la FPGA donde corre la emulación, y recibe datos desde la misma. Los datos que se envían por parte del usuario hacia la FPGA son:

- Datos de configuración del convertidor *fullbridge*: el usuario dispone de cajas de texto desde donde puede establecer el valor del condensador, de la bobina, de la resistencia, de la tensión del generador, del valor inicial de tensión del condensador, de la corriente inicial que atraviesa la bobina, y de los parámetros *Ciclos MAX* y *Ciclos ON*, con los que se establecerá el ciclo de trabajo como se ilustró en la Figura 3-5. Además, se dispone de un botón *send*, para mandar finalmente estos datos a través del socket de comunicaciones TCP.

- Datos de configuración de *trigger*: el usuario establece las condiciones de trigger mediante los parámetros:

    - *Wave*: para seleccionar mediante un *combobox* qué onda/canal es sobre la que se desea armar un *trigger*.
    - *Trigger Mode*: sirve para establecer haciendo uso de un *combobox* la dirección en la que se disparará el *trigger*. Esta, puede ser en sentido ascendente (*rising*), descendente (*falling*) o en cualquiera de los dos sentidos (*both*).

    Adicionalmente, se tiene un botón dedicado, *trigger*, para enviar estos datos de configuración junto al tiempo de muestreo (*Time sampling*), dada en segundos.

En las siguientes líneas se procede a exponer la lógica del código a grandes rasgos.

La función *main* del programa se encarga de inicializar la matriz que contiene los vectores de muestras que se van a recibir. Estos son: tensión del condensador del convertidor (tensión de salida, *dataY[0]*), corriente que atraviesa la bobina (*dataY[1]*) y el vector de tiempos (*dataX).*

A continuación se lanza un hilo en el que se ejecuta el método *updateData()*, que mediante un capturador manual de eventos, a modo de semáforo, espera la orden de actualización de los datos que se muestran por pantalla. En caso afirmativo, acondiciona los datos y ejecuta la rutina de pintado. El panel gráfico se gestiona utilizando métodos de la librería muy potente llamada *scottplot* [26], que además implementa de forma transparente al programador, algunas funcionalidades útiles para el análisis de gráficos como el zoom automático y manual, deslizamiento sobre la gráfica, etc.

Por otro lado, se invoca al método *connect(string server, int port)*, de la clase *MySocket*. Con este método, se crea un *socket* TCP para la comunicación con el sistema de procesamiento de la FPGA. Toma como parámetros la dirección IP del servidor (en este caso, la FPGA), *server*, y un puerto de conexión, *port.* Inmediatamente, se lanza otro hilo en el correará el método *getMessage()*, que se dedica a escuchar si entran datos nuevos provenientes de la FPGA en el buffer de recepción del socket, mediante un objeto de la clase *NetworkStream*. Cuando llegan datos, se almacena toda la ráfaga en una variable, y se acondiciona la matriz mencionada anteriormente, para que aloje los tres vectores comentados en el orden correcto en el que se envían en origen. Una vez completado este proceso, se habilita la orden de pintado que está esperando *updateData*.

Finalmente, en lo que se refiere al envío de datos de configuración del *fullbridge* y del armado del osciloscopio digital, desde la aplicación de escritorio hacia la FPGA, la clase *MySocket*, tiene su correspondiente método *setMessage(byte[] sendData)*, donde *sendData*, es el buffer de bytes a enviar. Los datos de este buffer son los de configuración del convertidor o los del osciloscopio digital en función de si se pulsa el botón *send* o *trigger*, respectivamente. Las variables de las cajas de texto se actualizan en tiempo real por medio de *bindings*, y las del resto de elementos por medio de escuchadores de eventos (*event listeners*). En el momento en el que se pulsa un botón u otro, se rellena el buffer con el contenido de las cajas de texto o de los *combobox*, todos ellos precedidos de un valor para que la FPGA distinga qué tipo de configuración está recibiendo. Por simplicidad en el envío de los valores, todos se tratan como números en coma flotante.

Como se ha comentado, antes de seguir con la sección que trata sobre la integración del osciloscopio digital, se han hecho pruebas para depurar todo lo implementado hasta ahora. Se ha comprobado el correcto comportamiento del *fullbridge* y de la comunicación bidireccional entre la FPGA y la aplicación de escritorio.

Por un lado se ha desarrollado la aplicación de escritorio íntegramente, pues la procedencia de los datos que se reciben es irrelevante siempre y cuando lleguen en el formato adecuado. Por eso, esta parte es independiente de la integración del osciloscopio. En la Figura 4-8 se muestra el resultado visual de la aplicación de escritorio, que ha sido descrito en esta sección.

Por otro lado, se ha verificado la correcta funcionalidad del *fullbridge* y de las comunicaciones con dirección aplicación de escritorio a FPGA. Para ello, se comprueba que los parámetros de configuración del *fullbridge* son recibidos por la FPGA en el orden y formato correctos, y que, tras la inicialización del módulo y la escritura de esos parámetros, los resultados de la tensión de salida son los esperados. Se puede comprobar el estado de una variable directamente por consola, leyendo el valor en la posición de memoria que ocupa. Como ejemplo, en la Figura 4-7, se muestra el valor que se lee de v_C, ya en régimen permanente, de una ejecución en la que se busca alcanzar un valor de 100 V ($d = 75\%, Vg = 200\,V$, aplíquese la ecuación (10)). Para ello se utiliza el comando *devmem*, seguido de la dirección de memoria del parámetro v_C (realmente, v_C_debug). Sabiendo que la dirección base del módulo del *fullbridge* es la 0x43C00000, y la del parámetro v_C_debug tiene un offset de 0x48 respecto a la anterior, la posición que hay que leer es la 0x43C00048.

```
#define FULLBRIDGE_BASE_ADDR 0x43C00000
#define XFULLBRIDGE_CONFIG_ADDR_VC_DEBUG_DATA 0x48
```

```
root@FullBridge_plnx:~# devmem 0x43c00048
0x42C6FEF2
```

Figura 4-7: Comprobación del valor v_C mediante el comando *devmem*

El resultado que se muestra por pantalla está en formato IEEE-754, que es el estándar para números en coma flotante de precisión simple (32 bits). Este resultado, traducido a formato decimal resulta ser 99,49. El valor esperado es de 100 V, sin embargo, debido al rizado de la tensión provocado por la conmutación del sistema y la dinámica del convertidor, el resultado entra dentro del rango de rizado esperado.

Por último, se verifica que las comunicaciones en dirección FPGA a la aplicación de escritorio son correctas. Para ello, en la FPGA se crean un par de señales sinusoidales que jugarían el papel de las señales v_C e i_L y se desarrolla un código parecido (aunque más elemental) al que implementará la IP del osciloscopio. Esto implica la detección de una condición de *trigger* configurada desde la aplicación de escritorio, la preparación de los datos una vez cumplida esa condición y el envío de los datos hacia la aplicación de escritorio a través del socket TCP/IP. Estos datos serían los tres vectores ya mencionados: v_C, i_L y un vector de tiempos. Finalmente, se verifica en la aplicación de escritorio que se muestran los datos esperados en pantalla. En la Figura 4-8 se aprecian los dos ejemplos de señales sinusoidales que se quisieron representar.

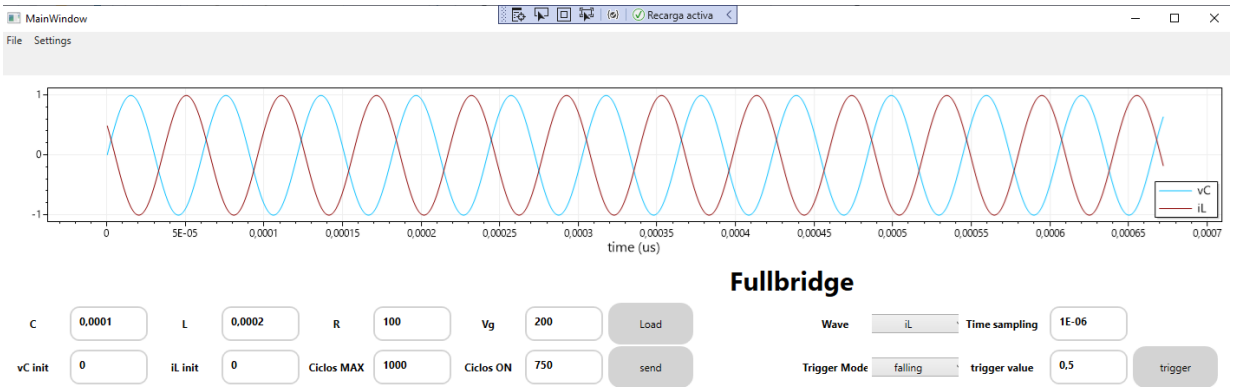

Figura 4-8: Entorno de diseño de la ventana principal, basado en XAML

4.5 Integración de un osciloscopio digital

Una vez verificado el funcionamiento del sistema básico, en esta última sección del capítulo 4, se explica qué modificaciones hay que realizar al diseño en cada una de sus partes para la integración de las IP del osciloscopio digital externo y del DMA reflejados en la Figura 4-1 y alcanzar así el objetivo final de este TFM.

Gracias a la integración de la IP del osciloscopio digital y de un controlador DMA, es posible visualizar la tensión de salida del convertidor y la corriente que atraviesa la bobina con baja latencia. Además de no ser por el uso del DMA, la cualidad de baja latencia se perdería, volviendo menos interesantes añadidos del estilo del osciloscopio digital que se ha integrado.

Para entender el funcionamiento de la IP del osciloscopio digital, se representa el diagrama de la Figura 4-9. Se puede comprobar que pasa por los estados de un osciloscopio convencional.

- Se parte de un estado de reposo a la espera de una petición de captura.

- Una vez se recibe una petición de captura se inicia la transacción mediante el armado del *trigger* con la configuración que se especifica desde la aplicación de escritorio.

- Cuando se cumple la condición de *trigger* sobre la señal seleccionada, se comienzan a capturar de las señales muestras hasta rellenar las señales que se representan: v_C, i_L y un vector de tiempos, t. El tamaño del buffer en esta aplicación es de 8000 muestras.

- Una vez se rellenen las 8000 muestras de cada una de las variables anteriores, se envían los datos hacia el DMA, el cual se encarga de escribir esos datos mediante un acceso directo a la memoria sin la intervención del procesador. De esta manera el procesador puede proseguir ejecutando su pila de instrucciones sin interrupción para gestionar la escritura de los datos de salida. Finaliza la transacción.

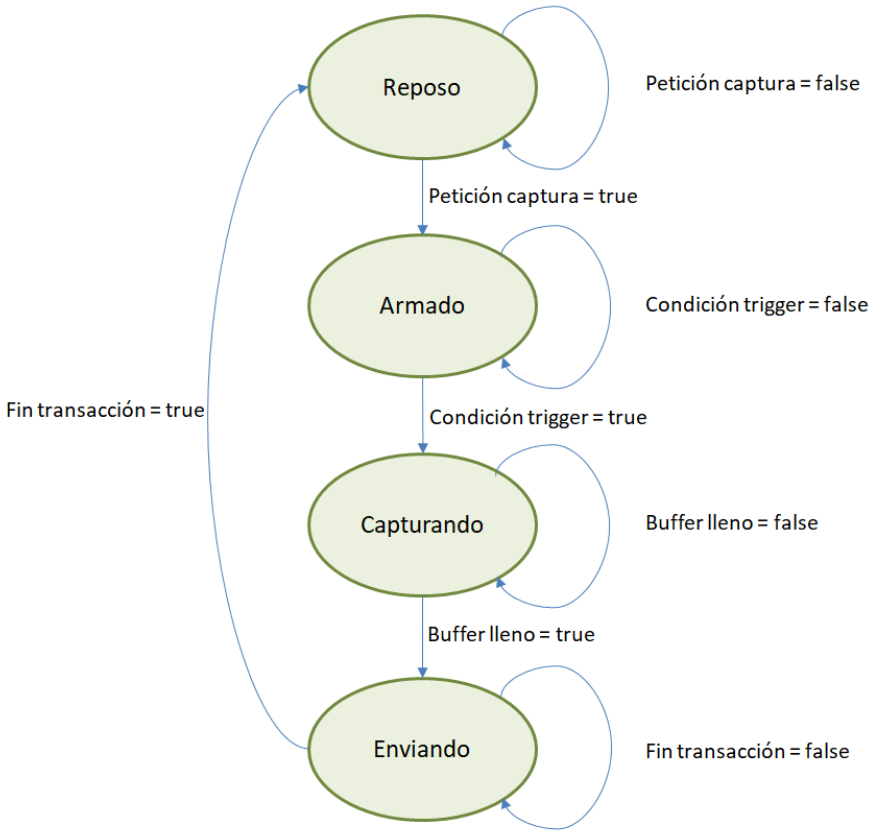

Figura 4-9: Máquina de estados que controla el osciloscopio

En lo que se refiere a la implementación, en la Figura 4-1 viene representado el conexionado entre el convertidor y el osciloscopio y entre el osciloscopio y el controlador DMA. Se hace notar que la IP del osciloscopio, permite conectar a su entrada hasta 8 canales. Sin embargo en este caso solo hacen falta tres (uno para v_C, otro para i_L y otro para la señal que establece una referencia de tiempos). Por último cabe destacar que la conexión del controlador DMA con el puerto de alto rendimiento (HP, *High Performance*) del PS se realiza por medio de un IP llamado *SmartConnect* [27], que permite conectar uno o más dispositivos AXI maestros mapeados en memoria con uno o más dispositivos AXI esclavos mapeados en memoria. Es decir, realiza un papel similar a un *hub* de conexiones.

Se procede a ejecutar la implementación del *Block Design* al igual que se hizo la primera vez, y se exporta el hardware para actualizarlo en Petalinux.

En Petalinux, se desea recompilar la distribución, incluyendo, entre, otros, el nuevo *device tree*, para que se reconozcan las nuevas IP introducidas. Adicionalmente, es necesario instalar el driver para manejar el controlador DMA y cambiar unos aspectos de la configuración para el manejo del mismo. En particular se ha utilizado el driver *xilinx_axidma* [28], que permite manejar el controlador DMA desde el espacio de usuario.

- Se lee el nuevo hardware exportado desde Vivado.

- Creación del driver *xilinx_axidma* (Figura 4-10).

```
SUMMARY = "Recipe for build an external xilinx-axidma Linux kernel module"
SECTION = "PETALINUX/modules"
LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://COPYING;md5=12f884d2ae1ff87c09e5b7ccc2c4ca7e"


inherit module


SRC_URI = "file://Makefile \
file://axi_dma.c \
file://axidma_chrdev.c \
file://axidma_dma.c \
file://axidma_of.c \
file://axidma.h \
file://axidma_ioctl.h \
file://COPYING \
"


S = "${WORKDIR}"


# The inherit of module.bbclass will automatically name module packages with
# "kernel-module-" prefix as required by the oe-core build environment.
```

Figura 4-10: Creación del driver *xiinx_axidma*

- Se deben compilar las librerías del módulo DMA importado, modificando para ello el *makefile* (script de compilación) que se genera automáticamente.

- El uso del DMA requiere habilitar el CMA (*Contiguous Memory Allocation*), bien accediendo mediante el comando petalinux-config –c kernel a las opciones para habilitar esta característica, o bien modificando manualmente el fichero .config, con la configuración del kernel. El CMA es una característica que hace que los *buffers* de memoria del DMA sean bloques contiguos en memoria, lo que permite velocidades de transferencia mucho mayores que si esos *buffers* de memoria estuvieran fragmentados.

```
CONFIG_CMA=y
CONFIG_DMA_CMA=y
CONFIG_XILINX_DMAENGINES=y
CONFIG_XILINX_AXIDMA=y
CONFIG_XILINX_AXIVDMA=y
CONFIG_DMA_SHARED_BUFFER=y
```

- El DMA debe estar reflejado en un nodo del *device-tree* del Sistema Operativo, por lo que hay que incluirlo. Para ello se incluye en el fichero de descripción del *device-tree*. Se hace notar en el código siguiente, que es importante que los *id* de

los canales de transmisión y recepción del DMA deben ser únicos, y por tanto, distintos entre sí.

```
axi_dma_0: dma@40400000 {
        #dma-cells = <1>;
        clock-names = "s_axi_lite_aclk", "m_axi_sg_aclk",
"m_axi_mm2s_aclk", "m_axi_s2mm_aclk";
        clocks = <&clkc 15>, <&clkc 15>, <&clkc 15>, <&clkc 15>;
        compatible = "xlnx,axi-dma-7.1", "xlnx,axi-dma-1.00.a";
        reg = <0x40400000 0x10000>;
        xlnx,addrwidth = <0x20>;
        xlnx,include-sg ;
        xlnx,sg-length-width = <0x17>;
        dma-channel@40400000 {
                compatible = "xlnx,axi-dma-mm2s-channel";
                dma-channels = <0x1>;
                xlnx,datawidth = <0x20>;
                xlnx,device-id = <0x0>;
        };
        dma-channel@40400030 {
                compatible = "xlnx,axi-dma-s2mm-channel";
                dma-channels = <0x1>;
                xlnx,datawidth = <0x20>;
                xlnx,device-id = <0x1>;
                xlnx,include-dre ;
        };
};
```

De igual manera, el driver requiere también de un nodo en el *device* tree. Cabe destacar que en el parámetro *dmas* es preciso verificar que vengan reflejados los dos canales del controlador DMA (*axi_dma_0*), distinguiéndose por un ‘0’ o un ‘1’. Debido a un *bug* del sistema, ambos canales vienen identificados con un ‘0’ por defecto. Esto significa que tanto para transmisión como para recepción se estará utilizando únicamente el canal ‘0’, lo cual es erróneo. Asimismo, a estos canales se les establece un nombre descriptivo, que en este caso son *tx_channel*, y *rx_channel*, respectivamente.

```
axidma_chrdev: axidma_chrdev@0 {
        compatible = "xlnx,axidma-chrdev";
        dmas = <&axi_dma_0 0 &axi_dma_0 1>;
        dma-names = "tx_channel", "rx_channel";
        };
```

Esta información debería crearse por defecto, pero debido a un *bug* del SO, hay que asegurar estos detalles manualmente.

Adicionalmente, en ese mismo fichero, se especifican unos parámetros de arranque:

```
bootargs = "console=ttyPS0,115200 earlyprintk cma=25M";
```

Donde destaca que se asigna un espacio de memoria contigua de 25 Mbytes, pues por defecto este espacio es bastante inferior e insuficiente para poder almacenar una trama grande capturada por el osciloscopio.

- Se recompila el sistema operativo de nuevo y se genera el archivo *booteable*.

Por otro lado, hay que modificar el código de programación de la sección 4.4 del procesador ARM en SDK una vez están integrados el osciloscopio digital y el DMA:

- Tal y como se explicó en la sección 4.3, ahora se añaden las nuevas funciones de lectura/escritura de parámetros de los nuevos módulos, así como las de inicialización.

- Tras la inicialización del controlador DMA se reserva la memoria necesaria para asegurar que hay espacio para todos los datos que envía el osciloscopio. Los datos son de tipo coma flotante, se muestran 8000 muestras por canal, y, aunque se usan solo 2 canales, el osciloscopio trabaja con los 8. En la Figura 4-11 se muestra cómo se llama a la función *axidma_malloc*, que reserva el espacio de memoria para el DMA. El parámetro *axidma_dev* es el puntero resultante de la inicialización del módulo DMA y que apunta a su dirección de memoria. El segundo parámetro es la cantidad de memoria que se desea reservar, explicado en las anteriores líneas de este mismo punto. Posteriormente, este puntero, así como otros parámetros relacionados con una transacción del DMA son asignados a una estructura, *trans*, que almacenará toda esta información. La definición del tipo de estructura se puede consultar en la Figura 4-12, donde se encuentran parámetros tales como el canal usado para enviar datos, la cantidad de datos que se envían, el buffer que contiene esos datos, etc.

```
buffer_dma = axidma_malloc(axidma_dev, 8000*8*sizeof(float));
{...}trans.output_buf = buffer_dma;
{...}
```

Figura 4-11: Reserva del espacio de memoria necesario para el controlador DMA

```
struct dma_transfer {
    int input_fd;           // The file descriptor for the input file
    int input_channel;      // The channel used to send the data
    int input_size;         // The amount of data to send
    void *input_buf;        // The buffer to hold the input data
    int output_fd;          // The file descriptor for the output file
    int output_channel;     // The channel used to receive the data
    int output_size;        // The amount of data to receive
    void *output_buf;       // The buffer to hold the output
};
```

Figura 4-12: Definición de una estructura adecuada para el transporte de la información que rodea a una transacción del DMA

- En el hilo secundario cuya función era la de escuchar nuevos datos de configuración que llegasen al buffer de recepción, se modifica el código para que cuando lleguen datos de configuración del *trigger*, se escriban en los parámetros de

la IP del osciloscopio digital. Seguidamente, se le indica al osciloscopio que esté listo para una nueva captura, y se prepara al DMA para realizar una nueva transacción. En cuanto el DMA recibe datos del osciloscopio, se produce la transacción de datos a través de la interfaz AXI. Cuando finaliza, se envían los datos a la aplicación de escritorio a través del socket. En la Figura 4-13, se muestran las partes principales de la función de preparación de transacción del DMA, *iniciar_trama_dma*. En primer lugar se llama a la función *axidma_stop_transfer*, la cual detiene cualquier transacción que se pudiera estar llevando a cabo en el canal del DMA al que se refiere *trans.output_channel*, que es el canal que se creó anteriormente para la transmisión de datos. En segundo lugar, con *axidma_set_callback*, se establece una función de *callback* (*dma_callback*). Esta función se ejecuta de forma asíncrona en el momento en el que finaliza una transacción, en este caso, en el canal *trans.output_channel*. En este caso, *dma_callback*, advierte de que la transacción se ha completado con éxito y acomete la escritura de los datos sobre el socket TCP/IP que llegarán a la aplicación de escritorio. Finalmente, se ejecuta la función *transfer_file*, que internamente llama a *axidma_oneway_transfer*, que es quien efectúa propiamente la transacción DMA.

```
void iniciar_trama_dma()
{
            {...}
            axidma_stop_transfer(axidma_dev, trans.output_channel );
            axidma_set_callback(axidma_dev, trans.output_channel, dma_callback,
&trans);
            // Transfer the file over the AXI DMA
            transfer_file(axidma_dev, &trans, output_path);
   }
```

Figura 4-13: Función de preparación de inicio de una transacción DMA

Para la consulta de todos los detalles que engloban a este proceso, refiérase al Anexo B.

De la aplicación de escritorio no hay que modificar nada, pues le es transparente cómo sea la llegada de datos, siempre y cuando lleguen en el mismo orden y formato.

Finalmente, si no se hubiera ordenado el autoarranque del módulo *xilinx*_axidma, se debería insertar manualmente desde una consola ejecutando el comando *insmod*. Para ello, solo faltaría iniciar una sesión con un cliente que permita establecer una comunicación para emular la *Shell* del Sistema Operativo e insertar manualmente el módulo del controlador DMA, que se encuentra en el *kernel*.

En la Figura 4-14, se demuestra cómo se realiza este último paso, y cómo, efectivamente, se detecta con éxito un periférico DMA con un canal de lectura y otro de escritura. . Sin embargo, solo se utiliza el canal que pasa los datos del osciloscopio a la memoria DDR. El comando que se utiliza es:

```
insmod/lib/modules/4.14.0-xilinx-v2018.3/extra/xilinx-axidma.ko
```

Siendo xilinx-axidma.ko, el nombre que se le ha dado al módulo DMA.

```
root@FullBridge_plnx:~# insmod /lib/modules/4.14.0-xilinx-v2018.3/extra/xilinx-axidma.ko

xilinx_axidma: loading out-of-tree module taints kernel.
axidma: axidma_dma.c: axidma_dma_init: 718: DMA: Found 1 transmit channels and 1 receive
 channels.
axidma: axidma_dma.c: axidma_dma_init: 720: VDMA: Found 0 transmit channels and 0 receiv
e channels.
root@FullBridge_plnx:~#
```

Figura 4-14: Inserción del módulo del DMA

5 Pruebas y resultados

Finalmente, ya se puede probar el sistema en pleno funcionamiento. A continuación, se hace una demostración del aspecto final del sistema completo en funcionamiento en las Figuras 5-1 a 5-6. En las propias figuras se observan los parámetros de configuración correspondientes a las capturas y se observa cómo los resultados finales de estabilización de la señal de salida del convertidor son congruentes con la ecuación (10) de la sección 3.2.

Para ahondar en las diferencias entre las imágenes, se recomienda centrar la atención en el espacio de configuración de parámetros, concretamente en los parámetros de configuración del *trigger*, en *Ciclos MAX*, en *Ciclos ON* y en *Vg*. De acuerdo a esto, de la Figura 5-1 se extrae que:

- La curva sobre la que actúa el *trigger* es la azul, que se corresponde con la tensión del condensador (*Wave = vC*).

- El *trigger* se activa cuando detecta un flanco de bajada en *vC* tomando como valor de comparación 75 V. (*Trigger Mode = falling, Trigger value = 75*).

- De acuerdo a la ecuación (10), se ve cómo *vC* tiende a un valor de 100 V en régimen permanente, siendo el ciclo de trabajo *d*, un 75%.

$$v_o = V_g \cdot (2 \cdot d - 1) = V_g \cdot \left(2 \cdot \frac{Ciclos\ ON}{Ciclos\ MAX} - 1\right) = 200 \cdot \left(2 \cdot \frac{750}{1000} - 1\right) = 100\ V$$

Otro ejemplo de funcionamiento es el de la Figura 5-2 se visualiza que:

- La curva sobre la que actúa el *trigger* es la roja, que se corresponde con la corriente de la bobina (*Wave = iL*).

- El *trigger* se activa cuando detecta un flanco de subida en *iL* tomando como valor de comparación 100 A. (*Trigger Mode = rising, Trigger value = 100*).

- De acuerdo a la ecuación (10), se ve cómo *vC* tiende a un valor de -250 V en régimen permanente, siendo el ciclo de trabajo *d*, un 75%.

$$v_o = V_g \cdot (2 \cdot d - 1) = V_g \cdot \left(2 \cdot \frac{Ciclos\ ON}{Ciclos\ MAX} - 1\right) = 500 \cdot \left(2 \cdot \frac{250}{1000} - 1\right) = -250\ V$$

Para complementar esta figura, se añade la Figura 5-3, donde los parámetros son los mismos a excepción del *Time sampling* (distancia entre muestras). En esta segunda figura, la distancia entre muestras de la captura del osciloscopio es mayor que en la Figura 5-1 (35 µs frente a 8µs), por lo que se observa cómo el tiempo de simulación es mayor. Además, se ve con más claridad el valor al que tiende a estabilizarse *vC*.

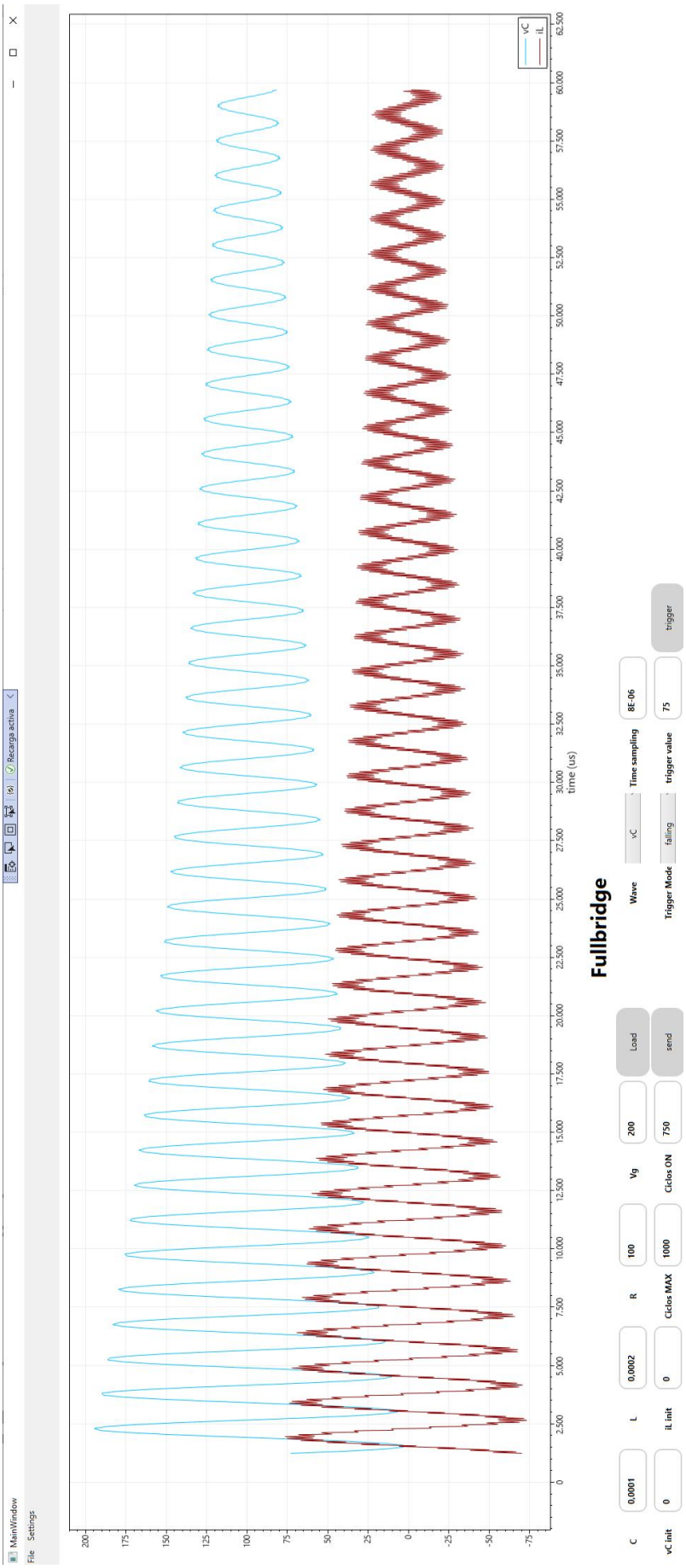

Figura 5-1: Ejemplo de funcionamiento 1

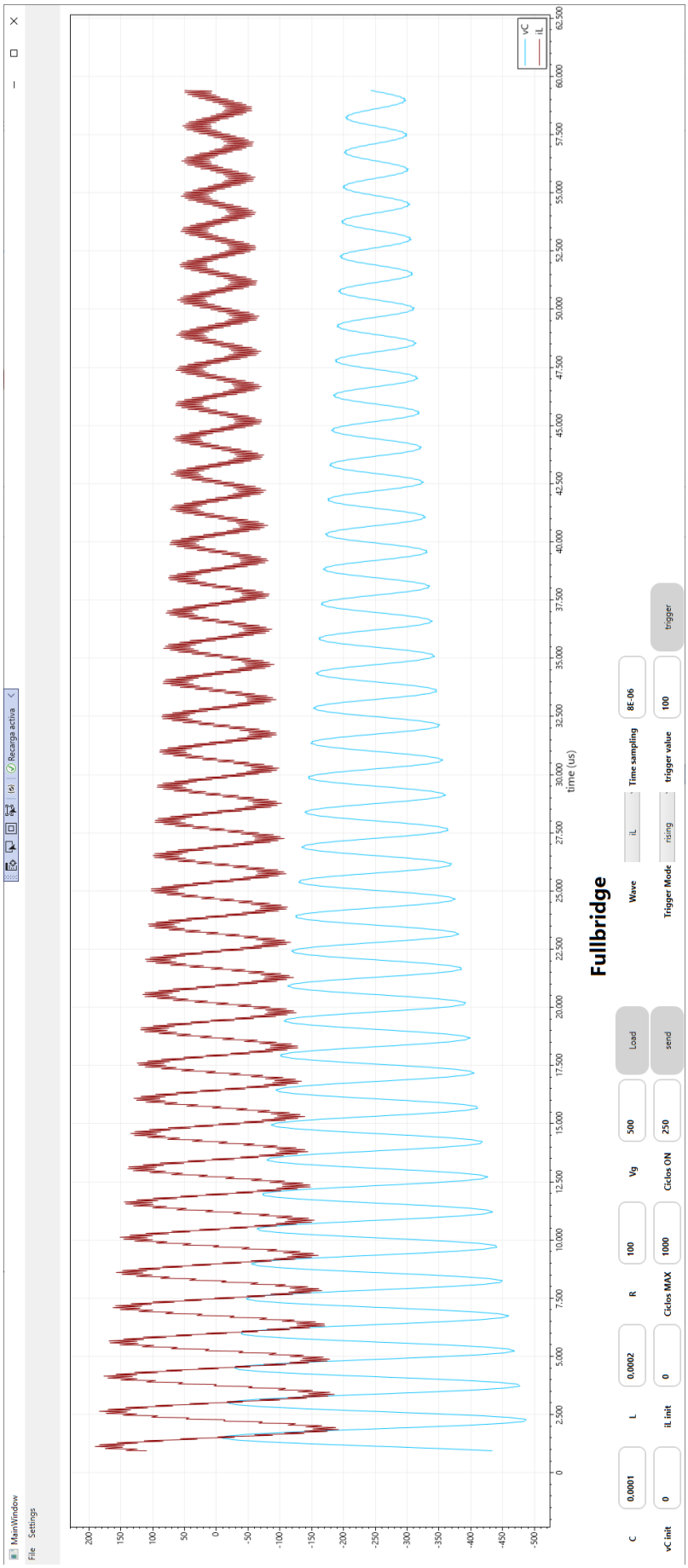

Figura 5-2: Ejemplo de funcionamiento 2

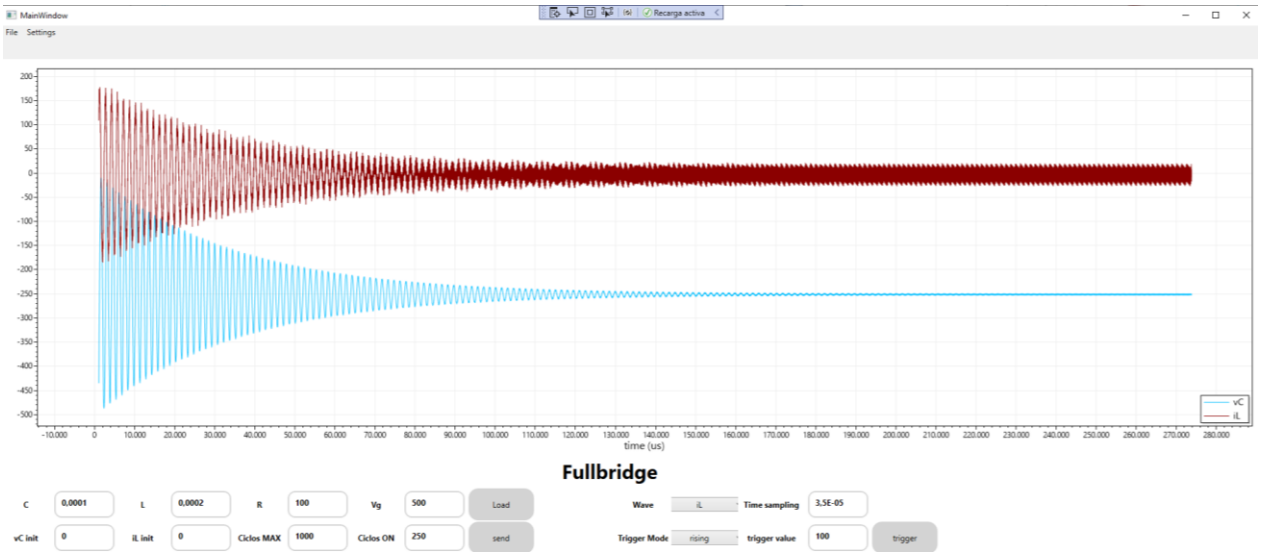

Figura 5-3: Ejemplo de funcionamiento 3

En la Figura 5-4 se demuestra el comportamiento de la aplicación con modificaciones en la planta respecto a los ejemplos anteriores. En concreto se ha aumentado la inductancia de la bobina desde 200 µH hasta 900 µH, la carga es de 12 Ω y la tensión de entrada 20 V. Se observa como al tener una bobina más grande, la corriente está mucho más filtrada y cómo el valor de tensión alcanzado, manteniendo el ciclo de trabajo en el control de los MOSFETs, es de 10 V.

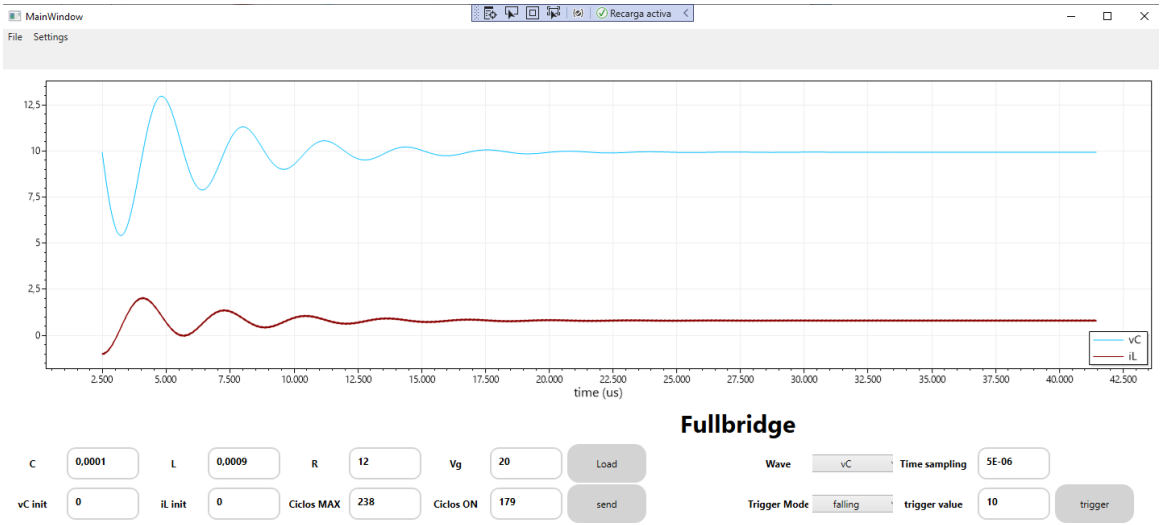

Figura 5-4: Ejemplo de funcionamiento 4

Analizando el rizado de la corriente en la Figura 5-5, se puede comprobar que su variación se corresponde con la frecuencia de conmutación:

$$T_{SW} = Ciclos\ MAX \cdot T_{CLK}$$

$$f_{SW} = \frac{1}{T_{SW}}$$

El ejemplo de la imagen está extraído con Ciclos MAX = 986 sabiendo que $T_{CLK} = 11{,}904\,MHz \approx 12MHz$, por lo que gráficamente se puede extraer que $T_{SW} \approx 83\,\mu s$, que se corresponden con una frecuencia de conmutación, f_{SW} de 12 kHz.

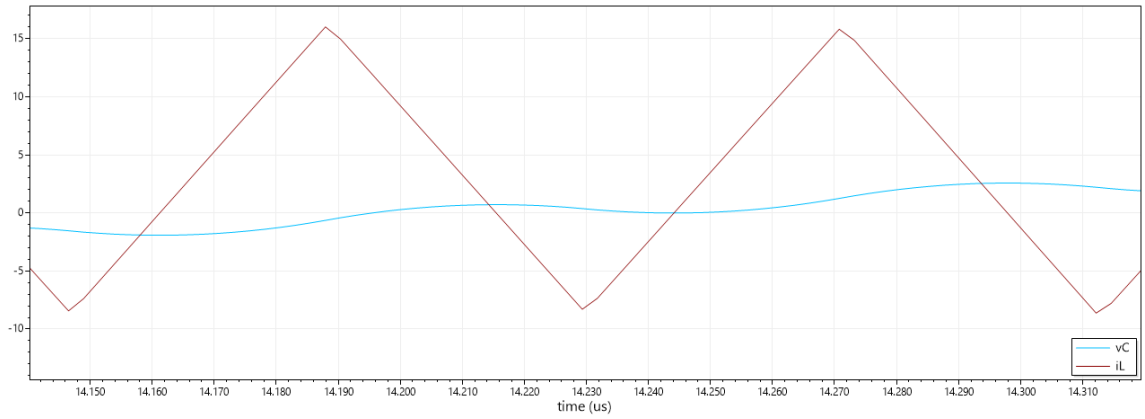

Figura 5-5: Rizado de la corriente de la bobina

Finalmente, en la Figura 5-6 se puede observar el detalle del arranque del convertidor.

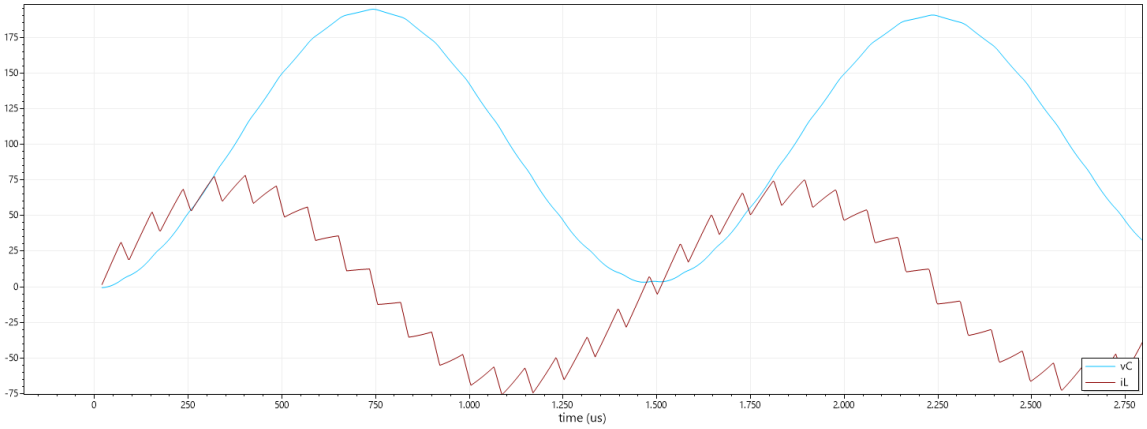

Figura 5-6: Arranque del convertidor

6 Conclusiones y trabajo futuro

6.1 Conclusiones

En este TFM se ha implementado un sistema de simulación HIL que permite depurar con mayor rapidez un sistema de control. Como demostración, se ha modelado en coma flotante y mediante síntesis de alto nivel (HLS) un convertidor de potencia conmutado *fullbridge*. El sistema se ha desarrollado sobre un SoC compuesto de una FPGA y un procesador ARM. La FPGA aporta una gran capacidad de procesamiento paralelo para reducir los pasos de simulación y dar la posibilidad de simular sistemas en tiempo real que conmuten a elevadas frecuencias. Para dotar de más versatilidad y escalabilidad al sistema, se ha instalado el Sistema Operativo Petalinux en el procesador ARM, con el que se ha facilitado la tarea de gestionar las comunicaciones entre la FPGA y un PC externo.

Adicionalmente al modelado de la planta, se ha integrado una IP de un sencillo osciloscopio digital con el que poder capturar los datos de salida del convertidor conmutado de acuerdo a unas condiciones de disparo. Esta información se traspasa de forma eficiente y con baja latencia a una memoria DDR de la FPGA mediante un DMA. Se ha desarrollado una aplicación de escritorio en el PC externo que se comunica con la FPGA para configurar el modelo y visualizar gráficamente los datos recibidos de la FPGA.

Se han realizado pruebas de integración del sistema para comprobar que la interactuación entre los distintos componentes del sistema es correcta y se ha verificado que el desempeño del modelo del convertidor *fullbridge* responde de manera fiel a su comportamiento teórico. Para ello se ha ilustrado los detalles de funcionamiento del sistema con diferentes valores de configuración del convertidor, variando tanto los valores de sus componentes discretos como el de su frecuencia de conmutación.

La solución llevada a cabo en este trabajo supone una alternativa frente a las soluciones tradicionales basadas en simuladores mixtos que pecan de ser lentos en términos de tiempo y que pierden la capacidad de realizar simulaciones en tiempo real.

6.2 Trabajo futuro

Del sistema implementado se puede sacar provecho para definir un trabajo futuro variado:

- En este TFM, se ha probado el modelo HIL en lazo abierto para ver su correcto funcionamiento. Sin embargo, como trabajo futuro se propone unir el modelo HIL implementado a un regulador digital externo, en configuración de lazo cerrado, ya que es la configuración real que utilizarán los usuarios potenciales del sistema diseñado.

- Podría mejorarse la IP del osciloscopio digital para dotarla de más funcionalidades propias de un osciloscopio real relacionadas con el *trigger* o con el procesamiento de la señal.

- Aprovechando que se ha instalado un Sistema Operativo basado en Linux en el procesador de la FPGA podría crearse una plataforma en la nube que permitiese interactuar con este sistema de forma remota.

Referencias

[1] A. De Castro. Aplicación del Control Digital Basado en Hardware Específico para Convertidores de Potencia Conmutados. *Tesis Doctoral*. 2003.

[2] 1. Zhang, S.; Liang, T.; Dinavahi, V. Machine Learning Building Blocks for Real-Time Emulation of Advanced Transport Power Systems. IEEE Open Journal of Power Electronics 2020, 1, 488–507 498. doi:10.1109/OJPEL.2020.3039117.

[3] Liang, T.; Liu, Q.; Dinavahi, V.R. Real-Time Hardware-in-the-Loop Emulation of High-Speed Rail Power System With SiC-Based Energy Conversion. IEEE Access 2020, 8, 122348–122359. doi:10.1109/ACCESS.2020.3006904.

[4] Mylonas, E.; Tzanis, N.; Birbas, M.; Birbas, A. An Automatic Design Framework for Real-Time Power System Simulators Supporting Smart Grid Applications. Electronics 2020, 9. doi:10.3390/electronics9020299.

[5] Samano-Ortega, V.; Padilla-Medina, A.; Bravo-Sanchez, M.; Rodriguez-Segura, E.; Jimenez-Garibay, A.; Martinez-Nolasco, J. Hardware in the Loop Platform for Testing Photovoltaic System Control. Applied Sciences 2020, 10. doi:10.3390/app10238690.

[6] Qi, C.; Gao, F.; Zhao, X.; Wang, Q.; Sun, Q. Distortion Compensation for a Robotic Hardware-In-The-Loop Contact Simulator. IEEE Transactions on Control Systems Technology 2018, 26, 1170–1179. doi:10.1109/TCST.2017.2709278.

[7] Li, Y.; Zhu, S.; Li, Y.; Lu, Q. Temperature Prediction and Thermal Boundary SimulationUsing Hardware-in-Loop Method for Permanent Magnet Synchronous Motors. IEEE/ASME Transactions on Mechatronics 2016, 21, 276–287. doi:10.1109/TMECH.2015.2443800.

[8] Ferraresi, C.; Maffiodo, D.; Franco, W.; Muscolo, G.G.; De Benedictis, C.; Paterna, M.; Pica, O.W.; Genovese, M.; Pacheco Quiñones, D.; Roatta, S.; Dvir, Z. Hardware-In-the-Loop Equipment for the Development of an Automatic Perturbator for Clinical Evaluation of Human Balance Control. Applied Sciences 2020, 10. doi:10.3390/app10248886.

[9] Lu, D.; Ma, Y.; Yin, H.; Deng, Z.; Qi, J. Development and Validation of Electronic Stability Control System Algorithm Based on Tire Force Observation. Applied Sciences 2020, 10. doi: 529 10.3390/app10238741.

[10] B. Lu, X. Wu, H. Figueroa, and A. Monti, "A low-cost real-time hardware-in-the-loop testing approach of power electronics controls," IEEE Trans. Ind. Electron., vol. 54, no. 2, pp. 919–931, Apr. 2007. doi: 10.1109/TIE.2007.892253.

[11] Bai, H.; Liu, C.; Zhuo, S.; Ma, R.; Paire, D.; Gao, F. FPGA-Based Device-Level Electro-Thermal Modeling of Floating Interleaved Boost Converter for Fuel Cell Hardware-in-the-Loop Applications. IEEE Transactions on Industry Applications 2019, 55, 5300–5310. doi: 10.1109/TIA.2019.2918048.

[12] Liu, C.; Guo, X.; Ma, R.; Li, Z.; Gechter, F.; Gao, F. A System-Level FPGA-Based Hardware-in-the-Loop Test of High-Speed Train. IEEE Transactions on Transportation Electrification 2018, 4, 912–921. doi: 10.1109/TTE.2018.2866696.

[13] Davalos-Guzman, U.; Castañeda, C.E.; Aguilar-Lobo, L.M.; Ochoa-Ruiz, G. Design and Implementation of a Real Time Control System for a 2DOF Robot Based on Recurrent High Order Neural Network Using a Hardware in the Loop Architecture. Applied Sciences 2021, 11. doi: 10.3390/app11031154.

[14] Lin N, Shi B, Dinavahi V. Non-linear behavioural modelling of device-level transients for complex power electronic converter circuit hardware realisation on fpga. IET Power Electron 2018;11(9):1566–74. https://doi.org/10.1049/ietpel. 2017.0212.
[15] https://www.opal-rt.com/. Última visita: 13/06/2021.
[16] https://www.dspace.com/en/pub/home.cfm. Última visita: 13/06/2021.
[17] https://www.typhoon-hil.com/. Última visita: 13/06/2021.
[18] Martínez-García MS, de Castro A, Sanchez A, Garrido. J, Word length selection method for HIL power converter models, International Journal of Electrical Power & Energy Systems, Volume 129, 2021, 106721, ISSN 0142-0615, https://doi.org/10.1016/j.ijepes.2020.106721.
[19] A. Sanchez, A. de Castro, and J. Garrido, "A comparison of simulation and hardware-in-the- loop alternatives for digital control of power converters," IEEE Trans. Ind. Informat., vol. 8, no. 3, pp. 491–500, Aug. 2012. doi: 10.1109/TII.2012.2192281.
[20] Zamiri E, Sanchez A, Yushkova M, Martínez-García MS, de Castro A. Comparison of Different Design Alternatives for Hardware-in-the-Loop of Power Converters. Electronics. 2021; 10(8):926. https://doi.org/10.3390/electronics10080926
[21] Zamiri E, Sanchez A, de Castro A, Martínez-García MS. Comparison of Power Converter Models with Losses for Hardware-in-the-Loop Using Different Numerical Formats. Electronics. 2019; 8(11):1255. https://doi.org/10.3390/electronics8111255
[22] William J. Palm III (2010). System Dynamics (2nd ed.). Boston: McGraw-Hill. p. 225. ISBN 978-0-07-126779-3.
[23] J. C. Butcher, Numerical Methods for Ordinary Differential Equations. New York, NY, USA: Wiley, 2016.
[24] M. Yushkova, A. Sanchez, A. de Castro, "Strategies for choosing an appropriate numerical method for FPGA-based HIL", International Journal of Electrical Power & Energy Systems, vol. 132, Nov. 2021. doi: 10.1016/j.ijepes.2021.107186
[25] https://developer.arm.com/architectures/system-architectures/amba. Última visita: 13/06/2021.
[26] https://github.com/ScottPlot/ScottPlot. Última visita: 13/06/2021.
[27] https://www.xilinx.com/products/intellectual-property/smartconnect.html. Última visita: 13/06/2021.
[28] https://github.com/bperez77/xilinx_axidma. Última visita: 13/06/2021.

Glosario

API	Application Programming Interface
TCP/IP	Transmission Control Protocol/Internet Protocol
IP	Intellectual Property
FPGA	Field Programmable Gate Array
HLS	High Level Synthesis
AXI	Advanced eXtensible Interface
AMBA	Advanced Microcontroller Bus Architecture
HIL	Hardware in the Loop

Anexos

A Código HLS

Código *fullbridge*

```
#include "fullbridge.h"

void fullbridge(bool load, float vc_init, float il_init, float vg, float ir,
bool mosfet_Q1, bool mosfet_Q2, bool mosfet_Q3, bool mosfet_Q4, float dtL,
float dtC, float *vc, float *il, float *numMuestra, float * vc_debug, float
*il_debug){
//#pragma HLS interface ap_none port=vg
#pragma HLS interface ap_none port=mosfet_Q1
#pragma HLS interface ap_none port=mosfet_Q2
#pragma HLS interface ap_none port=mosfet_Q3
#pragma HLS interface ap_none port=mosfet_Q4
#pragma HLS interface ap_none port=vc
#pragma HLS interface ap_none port=il
#pragma HLS interface ap_none port=numMuestra

#pragma HLS interface ap_ctrl_none port=return

#pragma HLS interface s_axilite port=load bundle=config
#pragma HLS interface s_axilite port=vc_init bundle=config
#pragma HLS interface s_axilite port=il_init bundle=config
//#pragma HLS interface s_axilite port=Rinv bundle=config
#pragma HLS interface s_axilite port=ir bundle=config
#pragma HLS interface s_axilite port=dtL bundle=config
#pragma HLS interface s_axilite port=dtC bundle=config
#pragma HLS interface s_axilite port=vc_debug bundle=config
#pragma HLS interface s_axilite port=il_debug bundle=config
#pragma HLS interface s_axilite port=vg bundle=config

#pragma HLS latency min=0 max=0

	static float numMuestra_float=0;
	static float vc_int, il_int;
	float vl, ic;
	static bool load_anterior = true;


	if (load_anterior != load)
		numMuestra_float=0.0;

	load_anterior = load;

	if (load == true)
	{
		il_int = il_init;
		vc_int = vc_init;
	}
	else
	{
		if ((mosfet_Q1 == true && mosfet_Q3 == true) && (mosfet_Q2 == false
&& mosfet_Q4==false))
```

```
            {
                        vl = vg - vc_int;
                        ic = il_int - ir;
            }
            else if ((mosfet_Q1 == false && mosfet_Q3 == false) && (mosfet_Q2
== true && mosfet_Q4==true))
            {
                        vl = -vg - vc_int;
                        ic = il_int - ir;
            }
            else if (mosfet_Q1 == false && mosfet_Q2 == false && mosfet_Q3 ==
false && mosfet_Q4 == false)
            {
                        if (il_int > 0)
                        {
                                    vl = -vg - vc_int;
                                    ic = il_int - ir;
                        }
                        else
                        {
                                    vl = vg - vc_int;
                                    ic = il_int - ir;
                        }
            }
            else if (mosfet_Q1 == true || mosfet_Q3 ==true)
            {
                        if (il_int > 0)
                        {
                                    vl = - vc_int;
                                    ic = il_int - ir;
                        }
                        else
                        {
                                    vl = vg - vc_int;
                                    ic = il_int - ir;
                        }
            }
            else if (mosfet_Q2 == true || mosfet_Q4 ==true)
            {
                        if (il_int > 0)
                        {
                                    vl = -vg - vc_int;
                                    ic = il_int - ir;
                        }
                        else
                        {
                                    vl = - vc_int;
                                    ic = il_int - ir;
                        }
            }


            il_int = il_int + vl * dtL;
            vc_int = vc_int + ic * dtC;
      }

      numMuestra_float++;
      *vc = vc_int;
      *il = il_int;
```

```
	*vc_debug = vc_int;
	*il_debug = il_int;
	*numMuestra = numMuestra_float;


}
```

Código *MOSFETs*

```
#include "PWM.h"

void PWM(int ciclosMAX, int ciclosON, bool encendido, bool *salida_Q1, bool
*salida_Q2, bool *salida_Q3, bool *salida_Q4){
#pragma HLS interface ap_none port=salida_Q1
#pragma HLS interface ap_none port=salida_Q2
#pragma HLS interface ap_none port=salida_Q3
#pragma HLS interface ap_none port=salida_Q4

#pragma HLS interface s_axilite port=ciclosMAX bundle=config
#pragma HLS interface s_axilite port=ciclosON bundle=config
#pragma HLS interface s_axilite port=encendido bundle=config

#pragma HLS latency min=0 max=0
#pragma HLS interface ap_ctrl_none port=return

	static int ciclosActuales=0;

	if (encendido == false)
	{
		*salida_Q1 = false;
		*salida_Q3 = false;

		*salida_Q2 = false;
		*salida_Q4 = false;
		ciclosActuales = 0;
	}
	else
	{
		if (ciclosActuales < ciclosON)
		{
			*salida_Q1 = true;
			*salida_Q3 = true;

			*salida_Q2 = false;
			*salida_Q4 = false;
		}
		else
		{
			*salida_Q1 = false;
			*salida_Q3 = false;

			*salida_Q2 = true;
			*salida_Q4 = true;
		}
		if (ciclosActuales < ciclosMAX)
			ciclosActuales++;
```

```
                else
                        ciclosActuales = 0;
        }
}
```

B Código SDK

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
#include <sys/mman.h>
#include <stdbool.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <math.h>
#include "axidma_ioctl.h"
#include "libaxidma.h"
#include <errno.h>

#define PI 3.14159265


#define PWM_BASE_ADDR   0x43C10000
#define FULLBRIDGE_BASE_ADDR 0x43C00000
#define OSCILO_BASE_ADDR 0x43C20000

//PWM Control IP memory directions
#define XPWM_CONFIG_ADDR_CICLOSMAX_DATA 0x10
#define XPWM_CONFIG_ADDR_CICLOSON_DATA  0x18
#define XPWM_CONFIG_ADDR_ENCENDIDO_DATA 0x20

//Oscilloscope IP memory directions
#define XOSCILO_CONFIG_ADDR_CONFIGSTART_DATA            0x10
#define XOSCILO_CONFIG_ADDR_RELOAD_DATA                 0x18
#define XOSCILO_CONFIG_ADDR_CONFIGSTATUS_DATA           0x20
#define XOSCILO_CONFIG_ADDR_TAMCAPTURA_DATA             0x28
#define XOSCILO_CONFIG_ADDR_NUMPETICIONES_DATA          0x30
#define XOSCILO_CONFIG_ADDR_NUMBYTESENVIADOS_DATA       0x38
#define XOSCILO_CONFIG_ADDR_MODOTRIGGER_DATA            0x40
#define XOSCILO_CONFIG_ADDR_CANALTRIGGER_DATA           0x48
#define XOSCILO_CONFIG_ADDR_VALORTRIGGER_DATA           0x50
#define XOSCILO_CONFIG_ADDR_DISTANCIAENTREMUESTRAS_DATA 0x58

//Fullbridge IP memory directions
#define XFULLBRIDGE_CONFIG_ADDR_LOAD_DATA     0x10
#define XFULLBRIDGE_CONFIG_ADDR_VC_INIT_DATA  0x18
#define XFULLBRIDGE_CONFIG_ADDR_IL_INIT_DATA  0x20
#define XFULLBRIDGE_CONFIG_ADDR_VG_DATA       0x28
#define XFULLBRIDGE_CONFIG_ADDR_IR_DATA       0x30
#define XFULLBRIDGE_CONFIG_ADDR_DTL_DATA      0x38
#define XFULLBRIDGE_CONFIG_ADDR_DTC_DATA      0x40
#define XFULLBRIDGE_CONFIG_ADDR_VC_DEBUG_DATA 0x48
#define XFULLBRIDGE_CONFIG_ADDR_VC_DEBUG_CTRL 0x4c
#define XFULLBRIDGE_CONFIG_ADDR_IL_DEBUG_DATA 0x50
#define XFULLBRIDGE_CONFIG_ADDR_IL_DEBUG_CTRL 0x54

void *ptr_PWM_mosfet, *ptr_fullbridge, *ptr_oscilo;
```

```
int new_data_osc_vC = 0;
int new_data_osc_iL = 0;

int newsockfd_w;

float wave_index = 0.0;
float trigger_mode = 0.0;
float trigger_mode_vC = 0.0;
float trigger_mode_iL = 0.0;
float iL_trigger = 0.2;
float vC_trigger = 0.5;
float trigger_value = 0.0;
float reload_t = 0.0;
float reload_vC = 0.0;
float reload_iL = 0.0;
float triggervC = 1.0;
float triggeriL = 1.0;
float triggered = 0.0;
float time_sampling = 1e-6;
float time_sampling_old = 1e-6;
int sampling_changed = 0;

int R = 1;

void * PWM_init()
{
	unsigned page_size = sysconf(_SC_PAGESIZE);
	unsigned page_addr_pwm;
	unsigned page_offset_pwm;
	int fd;
	page_addr_pwm = (PWM_BASE_ADDR & ~(page_size-1));
	page_offset_pwm = PWM_BASE_ADDR - page_addr_pwm;

	fd = open("/dev/mem",O_RDWR);
	if (fd < 1) {
		fprintf(stderr,"botones_init: Invalid mem file.\n");
		return NULL;
	}
	void *
ret=mmap(NULL,page_size,PROT_READ|PROT_WRITE,MAP_SHARED,fd,(PWM_BASE_ADDR &
~(page_size-1)));

	close(fd);

	return ret+page_offset_pwm;
}

void * oscilo_init()
{
	unsigned page_size = sysconf(_SC_PAGESIZE);
	unsigned page_addr_oscilo;
	unsigned page_offset_oscilo;
	int fd;
	page_addr_oscilo = (OSCILO_BASE_ADDR & ~(page_size-1));
	page_offset_oscilo = OSCILO_BASE_ADDR - page_addr_oscilo;

	fd = open("/dev/mem",O_RDWR);
	if (fd < 1) {
		fprintf(stderr,"botones_init: Invalid mem file.\n");
```

```
                return NULL;
        }
        void *
ret=mmap(NULL,page_size,PROT_READ|PROT_WRITE,MAP_SHARED,fd,(OSCILO_BASE_ADDR &
~(page_size-1)));

        close(fd);

        return ret+page_offset_oscilo;
}

void * fullbridge_init()
{
        unsigned page_size = sysconf(_SC_PAGESIZE);
        unsigned page_addr_fullbridge;
        unsigned page_offset_fullbridge;
        int fd;
        page_addr_fullbridge = (FULLBRIDGE_BASE_ADDR & ~(page_size-1));
        page_offset_fullbridge = FULLBRIDGE_BASE_ADDR - page_addr_fullbridge;

        fd = open("/dev/mem",O_RDWR);
        if (fd < 1) {
                fprintf(stderr,"botones_init: Invalid mem file.\n");
                return NULL;
        }
        void *
ret=mmap(NULL,page_size,PROT_READ|PROT_WRITE,MAP_SHARED,fd,(FULLBRIDGE_BASE_ADD
R & ~(page_size-1)));

        close(fd);

        return ret+page_offset_fullbridge;
}

int robust_write(int fd, char *buf, int buf_size)
{
    int bytes_remain, bytes_written;
    int buf_offset;

    // Read out the bytes into the buffer, accounting for EINTR
    bytes_remain = buf_size;
    while (true)
    {
        buf_offset = buf_size - bytes_remain;
        bytes_written = write(fd, buf + buf_offset, bytes_remain);
        bytes_remain = (bytes_written > 0) ? bytes_remain - bytes_written
                                           : bytes_remain;

        /* If we were interrupted by a signal, then repeat the write.
Otherwise,
         * if we encountered a different error or reached EOF then stop. */
        if (bytes_written < 0 && bytes_written != -EINTR) {
            return bytes_written;
        } else if (bytes_written == 0) {
            return buf_size - bytes_remain;
        }
    }

    // We should never reach here
```

```
    assert(false);
    return -EINVAL;
}

// OSCILLOSCOPE/DMA FUNCTIONS
// A convenient structure to carry information around about the transfer
struct dma_transfer {
    int input_fd;           // The file descriptor for the input file
    int input_channel;      // The channel used to send the data
    int input_size;         // The amount of data to send
    void *input_buf;        // The buffer to hold the input data
    int output_fd;          // The file descriptor for the output file
    int output_channel;     // The channel used to receive the data
    int output_size;        // The amount of data to receive
    void *output_buf;       // The buffer to hold the output
};


struct dma_transfer trans;
const array_t *rx_chans;
axidma_dev_t axidma_dev;
void * buffer_dma;

static int transfer_file(axidma_dev_t dev, struct dma_transfer *trans,
                         char *output_path)
{
    int rc;


    // Perform the transfer
    // Perform the main transaction
    rc = axidma_oneway_transfer(dev, trans->output_channel, trans->output_buf,
trans->output_size, false);

    if (rc < 0) {
        fprintf(stderr, "DMA read write transaction failed.\n");
     //   goto free_output_buf;
    }


ret:
    return rc;
}

// Function to write the oscilloscope output into a file
int file_dump(char* f, char *fdebug, int numMuestras, int numCanales)
{
       FILE *fp=NULL, *fp2=NULL;
       fp=fopen(f,"rb");
       if(!fp)
       {
              fprintf(stdout,"File %s does not exists\n", f);
              return -1;
       }

       fp2=fopen(fdebug,"w");
       if(!fp2)
       {
              fprintf(stdout,"File %s could not been created\n", f);
```

```
		return -1;
	}

	float buffer[1];

	if (numMuestras == 0 && numCanales == 0)
	{
		while( fread(buffer,sizeof(float),1,fp) >  0)
		{
			fprintf(stdout,"%f\n", buffer[0]);
			fprintf(fp2,"%f\n", buffer[0]);
		}
	}
	else
	{
		for(int i=0;i<numCanales;i++)
		{
			fprintf(fp2,"Canal: %d", i);
			for(int j=0; j<numMuestras; j++)
			{
				fread(buffer,sizeof(float),1,fp);
				fprintf(stdout,"%f\n", buffer[0]);
				fprintf(fp2,"%f\n", buffer[0]);
			}

		}

	}
	fclose(fp);
	return 0;
}

int id_captura=0, id_captura_ant=0;

char output_path[]="/mnt/captura.txt";
char output_path_debug[]="/mnt/capturaDebug.txt";

int tamCaptura = 8000,reload,modo_trigger,canal_trigger;
float valor_trigger,distanciaEntreMuestras;

void dma_callback(int channel_id, void *arg)
{

	fprintf(stdout,"DMA CALLBACK: Transferencia acabada correctamente. ID
actual: %d \n",id_captura);

	//int reload;
	struct dma_transfer *trans_int =  (struct dma_transfer*) arg;
	// Write the data to the output file
	//printf("Writing output data to `%s`.\n", (char*) output_path);

	//int  rc = robust_write(trans_int->output_fd, trans_int->output_buf,
	trans_int->output_size);


	write(newsockfd_w, trans_int->output_buf, 8000*4*3);
```

```
   //file_dump(output_path,output_path_debug,tamCaptura,3);
   //axidma_free(axidma_dev, trans_int->output_buf, trans_int->output_size);

   id_captura++;

}



void iniciar_trama_dma()
{
             printf("Configuring DMA.\n");

             axidma_stop_transfer(axidma_dev, trans.output_channel );
             trans.output_size = tamCaptura *8*4;

             axidma_set_callback(axidma_dev, trans.output_channel, dma_callback,
&trans);
             transfer_file(axidma_dev, &trans, output_path);
}

//READ/WRITE PWM CONTROL VALUES

int PWM_Get_CICLOSMAX(void *ptr)
{
return *((int *)(ptr+XPWM_CONFIG_ADDR_CICLOSMAX_DATA));
}

int PWM_Set_CICLOSMAX(void *ptr, int Data)
{
*((int *)(ptr+XPWM_CONFIG_ADDR_CICLOSMAX_DATA))= Data;
}

int PWM_Get_CICLOSON(void *ptr)
{
return *((int *)(ptr+XPWM_CONFIG_ADDR_CICLOSON_DATA));
}

int PWM_Set_CICLOSON(void *ptr, int Data)
{
*((int *)(ptr+XPWM_CONFIG_ADDR_CICLOSON_DATA))= Data;
}

bool PWM_Get_ENCENDIDO(void *ptr)
{
return *((bool *)(ptr+XPWM_CONFIG_ADDR_ENCENDIDO_DATA));
}

int PWM_Set_ENCENDIDO(void *ptr, bool Data)
{
*((bool *)(ptr+XPWM_CONFIG_ADDR_ENCENDIDO_DATA))= Data;
}

// READ/WRITE OSCILLOSCOPE VALUES

int Oscilo_Get_CONFIGSTART(void *ptr)
{
return *((int *)(ptr+XOSCILO_CONFIG_ADDR_CONFIGSTART_DATA));
}
```

```
int Oscilo_Set_CONFIGSTART(void *ptr, int Data)
{
*((int *)(ptr+XOSCILO_CONFIG_ADDR_CONFIGSTART_DATA))= Data;
}

int Oscilo_Get_RELOAD(void *ptr)
{
return *((int *)(ptr+XOSCILO_CONFIG_ADDR_RELOAD_DATA));
}

int Oscilo_Set_RELOAD(void *ptr, int Data)
{
*((int *)(ptr+XOSCILO_CONFIG_ADDR_RELOAD_DATA))= Data;
}

int Oscilo_Get_CONFIGSTATUS(void *ptr)
{
return *((int *)(ptr+XOSCILO_CONFIG_ADDR_CONFIGSTATUS_DATA));
}

int Oscilo_Set_CONFIGSTATUS(void *ptr, int Data)
{
*((int *)(ptr+XOSCILO_CONFIG_ADDR_CONFIGSTATUS_DATA))= Data;
}

int Oscilo_Get_TAMCAPTURA(void *ptr)
{
return *((int *)(ptr+XOSCILO_CONFIG_ADDR_TAMCAPTURA_DATA));
}

int Oscilo_Set_TAMCAPTURA(void *ptr, int Data)
{
*((int *)(ptr+XOSCILO_CONFIG_ADDR_TAMCAPTURA_DATA))= Data;
}

int Oscilo_Get_NUMPETICIONES(void *ptr)
{
return *((int *)(ptr+XOSCILO_CONFIG_ADDR_NUMPETICIONES_DATA));
}

int Oscilo_Set_NUMPETICIONES(void *ptr, int Data)
{
*((int *)(ptr+XOSCILO_CONFIG_ADDR_NUMPETICIONES_DATA))= Data;
}

int Oscilo_Get_NUMBYTESENVIADOS(void *ptr)
{
return *((int *)(ptr+XOSCILO_CONFIG_ADDR_NUMBYTESENVIADOS_DATA));
}

int Oscilo_Set_NUMBYTESENVIADOS(void *ptr, int Data)
{
*((int *)(ptr+XOSCILO_CONFIG_ADDR_NUMBYTESENVIADOS_DATA))= Data;
}

int Oscilo_Get_MODOTRIGGER(void *ptr)
{
return *((int *)(ptr+XOSCILO_CONFIG_ADDR_MODOTRIGGER_DATA));
```

```
}

int Oscilo_Set_MODOTRIGGER(void *ptr, int Data)
{
*((int *)(ptr+XOSCILO_CONFIG_ADDR_MODOTRIGGER_DATA))= Data;
}

int Oscilo_Get_CANALTRIGGER(void *ptr)
{
return *((int *)(ptr+XOSCILO_CONFIG_ADDR_CANALTRIGGER_DATA));
}

int Oscilo_Set_CANALTRIGGER(void *ptr, int Data)
{
*((int *)(ptr+XOSCILO_CONFIG_ADDR_CANALTRIGGER_DATA))= Data;
}

float Oscilo_Get_VALORTRIGGER(void *ptr)
{
return *((float *)(ptr+XOSCILO_CONFIG_ADDR_VALORTRIGGER_DATA));
}

int Oscilo_Set_VALORTRIGGER(void *ptr, float Data)
{
*((float *)(ptr+XOSCILO_CONFIG_ADDR_VALORTRIGGER_DATA))= Data;
}

float Oscilo_Get_DISTANCIAENTREMUESTRAS(void *ptr)
{
return *((float *)(ptr+XOSCILO_CONFIG_ADDR_DISTANCIAENTREMUESTRAS_DATA));
}

int Oscilo_Set_DISTANCIAENTREMUESTRAS(void *ptr, float Data)
{
*((float *)(ptr+XOSCILO_CONFIG_ADDR_DISTANCIAENTREMUESTRAS_DATA))= Data;
}


// READ/WRITE FULLBRIDGE VALUES

bool FullBridge_Get_LOAD(void *ptr)
{
return *((bool *)(ptr+XFULLBRIDGE_CONFIG_ADDR_LOAD_DATA));
}

int FullBridge_Set_LOAD(void *ptr, bool Data)
{
*((bool *)(ptr+XFULLBRIDGE_CONFIG_ADDR_LOAD_DATA))= Data;
}

float FullBridge_Get_VC_INIT(void *ptr)
{
return *((float *)(ptr+XFULLBRIDGE_CONFIG_ADDR_VC_INIT_DATA));
}

int FullBridge_Set_VC_INIT(void *ptr, float Data)
{
*((float *)(ptr+XFULLBRIDGE_CONFIG_ADDR_VC_INIT_DATA))= Data;
}
```

```
float FullBridge_Get_IL_INIT(void *ptr)
{
return *((float *)(ptr+XFULLBRIDGE_CONFIG_ADDR_IL_INIT_DATA));
}

int FullBridge_Set_IL_INIT(void *ptr, float Data)
{
*((float *)(ptr+XFULLBRIDGE_CONFIG_ADDR_IL_INIT_DATA))= Data;
}

float FullBridge_Get_IR(void *ptr)
{
return *((float *)(ptr+XFULLBRIDGE_CONFIG_ADDR_IR_DATA));
}

int FullBridge_Set_IR(void *ptr, float Data)
{
*((float *)(ptr+XFULLBRIDGE_CONFIG_ADDR_IR_DATA))= Data;
}

float FullBridge_Get_DTL(void *ptr)
{
return *((float *)(ptr+XFULLBRIDGE_CONFIG_ADDR_DTL_DATA));
}

int FullBridge_Set_DTL(void *ptr, float Data)
{
*((float *)(ptr+XFULLBRIDGE_CONFIG_ADDR_DTL_DATA))= Data;
}

float FullBridge_Get_DTC(void *ptr)
{
return *((float *)(ptr+XFULLBRIDGE_CONFIG_ADDR_DTC_DATA));
}

int FullBridge_Set_DTC(void *ptr, float Data)
{
*((float *)(ptr+XFULLBRIDGE_CONFIG_ADDR_DTC_DATA))= Data;
}

float FullBridge_Get_VC_DEBUG(void *ptr)
{
return *((float *)(ptr+XFULLBRIDGE_CONFIG_ADDR_VC_DEBUG_DATA));
}

int FullBridge_Set_VC_DEBUG(void *ptr, float Data)
{
*((float *)(ptr+XFULLBRIDGE_CONFIG_ADDR_VC_DEBUG_DATA))= Data;
}

float FullBridge_Get_IL_DEBUG(void *ptr)
{
return *((float *)(ptr+XFULLBRIDGE_CONFIG_ADDR_IL_DEBUG_DATA));
}

int FullBridge_Set_IL_DEBUG(void *ptr, float Data)
{
*((float *)(ptr+XFULLBRIDGE_CONFIG_ADDR_IL_DEBUG_DATA))= Data;
```

```
}

float FullBridge_Get_VG(void *ptr)
{
return *((float *)(ptr+XFULLBRIDGE_CONFIG_ADDR_VG_DATA));
}

int FullBridge_Set_VG(void *ptr, float Data)
{
*((float *)(ptr+XFULLBRIDGE_CONFIG_ADDR_VG_DATA))= Data;
}


//ERROR MESSAGE FUNCTION
void error(const char *msg)
{
    perror(msg);
    exit(1);
}

void enable_modules(){
      //PWM_Set_ENCENDIDO(ptr_PWM_mosfet, true);
      FullBridge_Set_LOAD(ptr_fullbridge, false);
}

void disable_modules(){
      //PWM_Set_ENCENDIDO(ptr_PWM_mosfet, false);
      FullBridge_Set_LOAD(ptr_fullbridge, true);
      //FullBridge_Set_IR(ptr_fullbridge, 0);
}

void * server_listen(void* p_newsockfd)
{
      int newsockfd = *(int*)(p_newsockfd);
      free(p_newsockfd);
      int n;
      char buffer_aux[4];

      float vc_init;
      float il_init;
      float ir;
      float dtl;
      float dtc;
      float vg;

      int ciclosmax;
      int cicloson;

      int args_data = 10;
      int args_osc = 7;

      float param[10];

      float f;
      int count;
      int args = args_data;


      int load;
```

```
	while(1){

		n = 1;
		count = 0;
		bzero(buffer_aux,4);

		do{

			n = read(newsockfd,buffer_aux,4); //Blocks until client has
written something
			if (n < 0) error("ERROR reading from socket");
			//printf("Here is the message: %s",buffer);



			memcpy(&f, buffer_aux, sizeof (float));

			if((count == 0) && (f == 1.0)){
				args = args_osc;

			}
			if((count == 0) && (f == 0.0)){
				args = args_data;

			}
			//if
			param[count] = f;
			count = count + 1;

		}while(count<args);

		if(args == args_osc){

			wave_index = param[1];
			trigger_mode = param[2];
			time_sampling = param[5];
			triggered = param[6];
			trigger_value = param[3];
			reload_t = param[4];

			sampling = round(time_sampling/Tclk);
			Oscilo_Set_RELOAD(ptr_oscilo, reload_t);
			Oscilo_Set_TAMCAPTURA(ptr_oscilo, tamCaptura);
			Oscilo_Set_MODOTRIGGER(ptr_oscilo, trigger_mode);
			Oscilo_Set_CANALTRIGGER(ptr_oscilo, wave_index);
			Oscilo_Set_DISTANCIAENTREMUESTRAS(ptr_oscilo, sampling);
			Oscilo_Set_VALORTRIGGER(ptr_oscilo, trigger_value);

			iniciar_trama_dma();

			Oscilo_Set_CONFIGSTART(ptr_oscilo, false);
			Oscilo_Set_CONFIGSTART(ptr_oscilo, true);

			if(time_sampling_old != time_sampling){
				sampling_changed = 1;
				time_sampling_old = time_sampling;
			}
		}
```

```
		else{
			vc_init = param[1];
			il_init = param[2];
			R = param[3];
			dtl = 50e-9/param[4];
			dtc = 50e-9/param[5];
			vg = param[6];

			ciclosmax = (int)param[7];
			cicloson = (int)param[8];
			load = (int)param[9];

			float check_vc_debug =
			FullBridge_Get_VC_DEBUG(ptr_fullbridge);

			FullBridge_Set_VC_INIT(ptr_fullbridge, vc_init);
			FullBridge_Set_IL_INIT(ptr_fullbridge, il_init);
			FullBridge_Set_IR(ptr_fullbridge, check_vc_debug/R);
			FullBridge_Set_DTL(ptr_fullbridge, dtl);
			FullBridge_Set_DTC(ptr_fullbridge, dtc);
			FullBridge_Set_VG(ptr_fullbridge, vg);

			PWM_Set_CICLOSMAX(ptr_PWM_mosfet, ciclosmax);
			PWM_Set_CICLOSON(ptr_PWM_mosfet, cicloson);

			if(load == 1)
				disable_modules();
			else
				enable_modules();
		}

	}
	//close(newsockfd);
	//close(sockfd);
}
//OPEN SOCKET AND LISTEN FUNCTION
int create_server()
{
	//sockfd y newsockfd son file descriptors
	//portno is the port where the server accepts conexions
	// Clilen is the client IP size. Necessary by the system call accept()
	//n es el valor que retornan las llamadas a read() y write(). Contiene el
numero de caracteres leidos o escritos


	int sockfd, newsockfd, portno, clilen, n;

	//serv_addr will have server IP direction
	//cli_addr will have client IP direction
	struct sockaddr_in serv_addr, cli_addr;
	char buffer[256]; //hacer float??
	char buffer_aux[4];

	sockfd = socket(AF_INET, SOCK_STREAM, 0);
	if (sockfd < 0)
	  error("ERROR opening socket");

	bzero((char *) &serv_addr, sizeof(serv_addr));
	portno = 10123;
```

```
	serv_addr.sin_family = AF_INET;
	serv_addr.sin_port = htons(portno); //convert host format to network
format
	serv_addr.sin_addr.s_addr = INADDR_ANY; //INADDR_ANY te coge la IP del
host


	if (bind(sockfd, (struct sockaddr *) &serv_addr,sizeof(serv_addr)) < 0)
	  error("ERROR on binding");

	listen(sockfd,5);

	clilen = sizeof(cli_addr);


	newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
	if (newsockfd < 0)
	  error("ERROR on accept");

	pthread_t t;
	int *p_newsockfd = malloc(sizeof(int));
	*p_newsockfd = newsockfd;
	pthread_create(&t, NULL, server_listen, p_newsockfd); //add flag -
lpthread on C/C++ Build -> Settings -> Tool Settings tab -> GCC C++ Linker ->
Miscellaneous

	return newsockfd;
}



int main(){

	int check_CICLOSMAX;
	int check_CICLOSON;
	bool check_ENCENDIDO;

	bool check_LOAD;

	int w; //write value


	bool isUnder_vC = false;
	bool isAbove_vC = false;
	bool triggered_sin = false;

	bool isUnder_iL = false;
	bool isAbove_iL = false;
	bool triggered_cos = false;

	bool written = false;

	newsockfd_w = create_server();

	float *seno, *coseno;
	int numMuestras = 8000;
```

```
   float Tclk = 1/50e6;
   int sampling;

   float bufferwrite[numMuestras*3];

   sampling = round(time_sampling/Tclk);



   /*for(int i=0; i<numMuestras*3; i++){
         if (i < numMuestras)
               bufferwrite[i] = static_seno[i];
         else if(i <numMuestras*2)
               bufferwrite[i] = static_coseno[i-numMuestras];
         else
               bufferwrite[i] = static_tiempo[i-numMuestras*2];
   }
    */

   //w = write(newsockfd_w, bufferwrite, sizeof(bufferwrite));

   //write(newsockfd_w, static_seno, sizeof(static_seno));
   //write(newsockfd_w, static_coseno, sizeof(static_coseno));
   //write(newsockfd_w, static_tiempo, sizeof(static_tiempo));
//w = write(newsockfd_w,"I got your message",18);
//if (w < 0) error("ERROR writing to socket");


//write(fd,buf,strlen(buf));
//write(fd,&ret_val,sizeof ret_val);

//float number = 123.45;
//send(sockfd, &number, sizeof(float),0);

//fprintf(stderr, "errrrrrrrrrr");

   float check_vc_init;
   float check_il_init;
   float check_ir;
   float check_dtl;
   float check_dtc;
   float check_vc_debug;
   float check_il_debug;

   //inicializo
   ptr_PWM_mosfet = PWM_init();
   ptr_fullbridge = fullbridge_init();
   ptr_oscilo = oscilo_init();

   if (ptr_oscilo == MAP_FAILED) {
         perror("mmap");
         fprintf(stderr, "No se pueden inicializar el oscilo.\n");
         return EXIT_FAILURE;
   }

   axidma_dev = axidma_init();
   if (axidma_dev == NULL) {
   fprintf(stderr, "Error: Failed to initialize the AXI DMA device.\n");
   return -1;
```

```
	}

	rx_chans = axidma_get_dma_rx(axidma_dev);
	if (rx_chans->len < 1) {
	fprintf(stderr, "Error: No receive channels were found.\n");
	return -1;
	}


	buffer_dma = axidma_malloc(axidma_dev, 8000*8*sizeof(float));

	memset(&trans, 0, sizeof(trans));
	trans.output_fd = open(output_path, O_WRONLY|O_CREAT|O_TRUNC,

S_IWUSR|S_IRUSR|S_IRGRP|S_IWGRP|S_IROTH);
	trans.output_channel = rx_chans->data[0];

	// If the output size was not specified by the user, set it to the
default
	trans.input_size = tamCaptura*sizeof(float)*8;
	if (trans.output_size == -1) {
		trans.output_size = trans.input_size;
	}


	trans.output_buf = buffer_dma;
	if (trans.output_buf == NULL) {
		fprintf(stderr, "Fatal error. DMA roto");
		//goto destroy_axidma;
	}

	disable_modules();

	check_vc_debug = FullBridge_Get_VC_DEBUG(ptr_fullbridge);
	check_il_debug = FullBridge_Get_IL_DEBUG(ptr_fullbridge);

	check_LOAD = FullBridge_Get_LOAD(ptr_fullbridge);

	check_vc_debug = FullBridge_Get_VC_DEBUG(ptr_fullbridge);
	check_il_debug = FullBridge_Get_IL_DEBUG(ptr_fullbridge);

	//Compruebo los valores que he establecido
	check_ENCENDIDO = PWM_Get_ENCENDIDO(ptr_PWM_mosfet);
	check_CICLOSMAX = PWM_Get_CICLOSMAX(ptr_PWM_mosfet);
	check_CICLOSON = PWM_Get_CICLOSON(ptr_PWM_mosfet);

	//compruebo los valores que he establecido
	check_vc_init = FullBridge_Get_VC_INIT(ptr_fullbridge);
	check_il_init = FullBridge_Get_IL_INIT(ptr_fullbridge);
	check_ir = FullBridge_Get_IR(ptr_fullbridge);
	check_dtl = FullBridge_Get_DTL(ptr_fullbridge);
	check_dtc = FullBridge_Get_DTC(ptr_fullbridge);

	enable_modules();
	PWM_Set_ENCENDIDO(ptr_PWM_mosfet, true);

	check_LOAD = FullBridge_Get_LOAD(ptr_fullbridge);
	check_ENCENDIDO = PWM_Get_ENCENDIDO(ptr_PWM_mosfet);
```

```
	while(1){

		check_vc_debug = FullBridge_Get_VC_DEBUG(ptr_fullbridge);
		FullBridge_Set_IR(ptr_fullbridge, check_vc_debug/100);

		if(reload_vC == 0 && triggered_sin == true){
			if(new_data_osc_vC == 1){
				triggered_sin = false;
				new_data_osc_vC = 0;
				written = false;
			}
		}
		else{

		}

		if(reload_iL == 0 && triggered_cos == true){
			if(new_data_osc_iL == 1){
				triggered_cos = false;
				new_data_osc_iL = 0;
				written = false;
			}
		}
		else{

		}



	}
	//checking values
	/*while(1){
		check_vc_debug = FullBridge_Get_VC_DEBUG(ptr_fullbridge);
		FullBridge_Set_IR(ptr_fullbridge, check_vc_debug/100);
		check_ir = FullBridge_Get_IR(ptr_fullbridge);
		check_il_debug = FullBridge_Get_IL_DEBUG(ptr_fullbridge);
	}*/
	//botones_status = botones_read(ptr_buttons);
	//leds_check = leds_write(ptr_led, led_value_write);

}
```